VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Jednoduchá interaktivní hra ilustrující práci s permutačními programy

Simple Interactive Game Illustrating Working with Permutation Programs

2019 Ondřej Chovanec

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Ondřej Chovanec**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Jednoduchá interaktivní hra ilustrující práci s permutačními programy
Simple Interactive Game Illustrating Working with Permutation Programs

Jazyk vypracování: čeština

Zásady pro vypracování:

Permutační programy jsou jednoduchým výpočetním modelem, kde příslušný program je popsán jako sekvence permutací, přičemž vstupní data (ve formě sekvence hodnot 0 a 1) umožňují vybrat jako některé permutace v této sekvenci vždy jednu z každé určené dvojice permutací. Výstup programu je pak dán celkovou permutací, která vznikne složením všech pevně daných permutací a všech permutací vybraných na základě vstupních dat.
Cílem této bakalářské práce je vytvořit jednoduchou logickou hru, která umožní interaktivní manipulaci s permutačními programy a kde cílem bude nalézt příklad vstupních dat, pro která bude dosažen požadovaný výstup. Vytvořená aplikace by měla poskytovat grafické uživatelské rozhraní pro vytváření, editování a testování permutačních programů.

1. Nastuduje problematiku permutačních programů.
2. Navrhněte vhodné uživatelské rozhraní pro práci s permutačními programy.
3. Implementujte program pro interaktivní práci s permutačními programy ve formě hry. Tento program by měl umožňovat interaktivní vytváření permutačních programů.
4. Vytvořte sadu konkrétních netriviálních permutačních programů, na kterých bude možné činnost vytvořeného programu otestovat.

Seznam doporučené odborné literatury:

[1] D. A. Barrington, Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1, Journal of Computer and System Sciences, Volume 38, Issue 1, pp. 150-164, Elsevier, 1989.
[2] U. Hertrampf, C. Lautemann, T. Schwentick, H. Vollmer, K.W. Wagner, On the power of polynomial time bit-reductions, in: Proc. Eighth Annual Structure in Complexity Theory Conference, pp. 200-207, IEEE Computer Society Press, 1993.

Další literatura podle pokynů vedoucího práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. Ing. Zdeněk Sawa, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2019

Ondřej Chovanec .....

Rád bych poděkoval panu doc. Ing. Zdeňku Sawovi, Ph.D. za jeho věcné připomínky, dobré rady a vstřícnost při konzultacích.

Abstrakt

Cílem této práce je vytvořit aplikaci, která umožní vytváření, editaci a testování permutačních programů prostřednictvím interaktivní hry. Jsou zde vysvětleny pojmy permutace a permutační program, kterým je nutno rozumět pro pochopení aplikace. Další část práce popisuje, jak lze permutační program implementovat. Hlavní částí je vytvoření jednoduché logické hry, která umožňuje interaktivní manipulaci s permutačními programy, kde cílem je nalézt příklad vstupních dat, pro která vrací permutační program požadovaný výstup. Je ukázáno uživatelské rozhraní této hry a jsou probrány zajímavosti a problémy jeho implementace. Na závěr je uveden seznam ukázkových her, na kterých lze aplikaci otestovat.

Klíčová slova: permutace, permutační program, hra, implementace, uživatelské rozhraní

Abstract

The aim of this thesis is to create an application that enables creation, editing and testing of permutation programs through interactive game. There are explained the terms of permutation and permutation program which must be understood for understanding the application. Another part of the work describes how the permutation program can be implemented. The main part is to create a simple logic game that allows interactive manipulation with permutation programs, where the aim is to find an example of input data for which the permutation program returns the desired output. It shows the user interface of this game and discusses the issues and problems of its implementation. In conclusion, there is a list of sample games where you can test the application.

Key Words: permutation, permutation program, game, implementation, user interface

Obsah

Seznam použitých zkratek a symbolů

UI	–	User Interface

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Cílem této práce bylo vytvoření jednoduché hry s permutačními programy. Úkolem hráče hry je najít vstup, pro který permutační program vrátí požadovaný výstup. Aplikace měla umožňovat nejen hraní hry pomocí interaktivní manipulace s programy, ale i editaci existujících a vytváření nových permutačních programů. Měla mít grafické uživatelské rozhraní poskytující jednoduchou a intuitivní práci s permutačními programy.

Provedu vás tedy postupně kroky, které vedly k jejímu vytvoření. Nejdřív si objasníme teorii kolem pojmů permutace a permutační program v Kapitole 2. Podíváme se, jak se permutace zapisují, skládají a jak s tím vším souvisí permutační program. Kapitola 3 obsahuje popis hry a struktury aplikace. Poté si ukážeme samotný návrh aplikace. V Kapitole 4 detailně vysvětlím implementaci tříd reprezentující permutační programy a popíšu funkcionalitu jejich jednotlivých vlastností a metod. Druhou částí programu je implementace uživatelského rozhraní. V Kapitole 5 si ho nejdřív popíšeme z pohledu uživatele a řekneme si, co všechno aplikace nabízí. Kapitola 6 přibližuje, jak jsou některé složitější funkcionality UI implementovány, jaké problémy přitom nastaly a jak byly vyřešeny. Vysvětlím použité postupy a techniky a jaké jsou další možné alternativy s jejich výhodami a nevýhodami. Nakonec v Kapitole 7 popíšu ukázkové hry, které jsem vytvořil k otestování funkčnosti aplikace.

2 Permutace

2.1 Definice

Základem našeho programu jsou permutační programy, které se skládají z **permutací**. Abychom mohli rozumět pojmu permutace, musíme si nejdřív vysvětlit pojmy **injekce**, **surjekce** a **bijekce**.

Definice 1 ***Injekce*** *je zobrazení $f : A \to B$, které ke každé dvojici různých vzorů $x_1 \neq x_2$ přiřazuje dva různé obrazy $f(x_1) \neq f(x_2)$. Potom říkáme, že toto zobrazení je prosté neboli* ***injektivní****. [1]*

Definice 2 ***Surjekce*** *je zobrazení $f : A \to B$, pro které platí, že ke každému $y \in B$ existuje $x \in A$ takové, že $y = f(x)$. Potom o zobrazení f říkáme, že je* ***na množinu*** *B neboli* ***surjektivní****. [1]*

Definice 3 *Je-li zobrazení $f : A \to B$ injektivní a surjektivní, pak se nazývá vzájemně jednoznačné nebo-li* ***bijektivní****. Vzájemně jednoznačnému zobrazení také říkáme* ***bijekce****. [1]*

Definice 4 *Bijektivní zobrazení množiny na sebe $\pi : A \to A$, nazýváme* ***permutace*** *na množině A. [1]*

Počet permutací množiny o velikosti n je roven $P(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$.

2.2 Zápis

Abychom byli schopni s permutacemi pracovat, musíme je umět zapisovat. Permutace budeme značit π. Mějme množinu $M = \{1, 2, 3, 4, 5\}$ a definujme si zobrazení $\pi : M \to M$, takové že platí $\pi(1) = 2$, $\pi(2) = 4$, $\pi(3) = 5$, $\pi(4) = 1$, $\pi(5) = 3$. Jelikož žádné dva různé vzory nemají stejný obraz a zároveň všechny prvky M jsou obsaženy v oboru hodnot π, který se značí $H(\pi)$, jedná se o bijekci množiny na sebe tedy permutaci. Máme několik možností jak zapsat permutaci. První možností je zapsat permutaci dvouřádkovou maticí:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 5 & 1 & 3 \end{pmatrix}$$

V prvním řádku jsou vzory a pod nimi jejich obrazy. Jelikož v této práci budeme pracovat s množinami typu $[1, n]$, kde $n \in N$, můžeme první řádek se vzory vypustit a zapisovat permutaci jednořádkově:

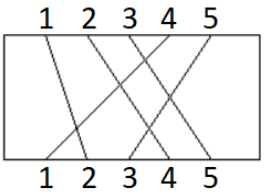
$$\pi = (2, 4, 5, 1, 3)$$

Píšeme tedy pouze obrazy a vzor nám určuje pozice obrazu v permutaci. Poslední možností je zápis permutace pomocí cyklů. Podíváme-li se na permutaci π můžeme si všimnout, že 1 se zobrazí na 2, 2 na 4 a 4 zpátky na 1. Vytvořil se tedy jakýsi cyklus $1 \to 2 \to 4 \to 1$. Jedná se

o cyklus délky 3. Můžeme najít i druhý cyklus $3 \to 5 \to 3$ délky 2. V případě, že by se prvek zobrazoval sám na sebe, jednalo by se o cyklus délky 1. Jelikož všechny prvky se objeví v právě jednom cyklu, jsou cykly navzájem disjunktní. Zápis pomocí cyklů permutace π potom vypadá takto:

$$\pi = (1, 2, 4)(3, 5)$$

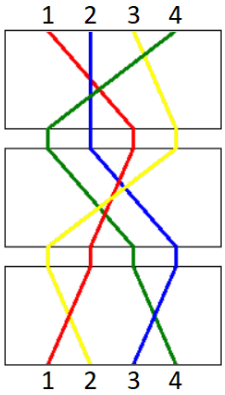
V cyklickém zápisu se obraz prvku objevuje napravo a vzor prvku nalevo od něj. Pokud by se prvek nacházel na konci cyklu jeho obrazem je první prvek v cyklu, tedy vzorem prvního prvku je prvek na konci cyklu. Pokud by se jednalo o cyklus délky jedna, znamená to, že prvek je sám sobě obrazem. Jelikož by mohlo dojít k záměně mezi jednořádkovým a cyklickým zápisem v případě, že se permutace skládá pouze z jednoho cyklu, hned teď si stanovíme pravidlo, že pokud nebude řečeno jinak, budeme v rozsahu této práce psát permutace jednořádkovým zápisem. Permutace budeme zobrazovat také graficky. Permutace $(2, 4, 5, 1, 3)$ značí, že první prvek se přesune na druhou pozici, druhý na čtvrtou pozici, třetí na pátou atd. Toto můžeme zobrazit pomocí čar, které zobrazují, jak si prvky vyměňují pozice, což je ukázáno na Obrázku 1.



Obrázek 1: Grafické zobrazení permutace $(2, 4, 5, 1, 3)$

2.3 Skládání

Zobrazení můžeme skládat. Permutace jsou zobrazení, tudíž je taky můžeme skládat. Obecně platí pro dvě zobrazení $g : A \to B$ a $f : B \to C$, že složené zobrazení $(f \circ g) : A \to C$ je takové zobrazení, že platí $(f \circ g)(x) = f(g(x))$ pro každé $x \in A$. Skládání zobrazení není komutativní $f \circ g \neq g \circ f$, avšak je asociativní $(f \circ g) \circ h = f \circ (g \circ h) = f \circ g \circ h$.[1] Nyní si ukážeme příklad skládání permutací. Mějme cyklicky zapsány permutace $\pi_1 = (1, 2)(3, 4)$, $\pi_2 = (1, 3, 2, 4)$ a $\pi_3 = (1, 3, 4)(2)$. Hledáme permutaci $\pi_4 = \pi_1 \circ \pi_2 \circ \pi_3$. Vezmeme jednotlivé čísla z množiny $[1, 4]$ a postupně zprava doleva skládáme jednotlivé permutace. Například pro číslo 3 nám vyjde $\pi_3(3) = 4 \to \pi_2(4) = 1 \to \pi_1(1) = 2$ tedy $\pi_4(3) = 2$. Pokud toto uděláme pro všechny čísla dostaneme $\pi_4 = (1)(2, 3)(4)$. Toto skládání lze zobrazit i graficky, jak je ukázáno na Obrázku 2. Permutace jsou shora dolu uspořádány, tak jak jsou postupně skládány. Pro nás je to tedy sekvence (π_3, π_2, π_1).



Obrázek 2: Grafické zobrazení skládání permutací

2.4 Permutační programy

Definice 5 ***Permutační program*** *je jednoduchý výpočetní model, kde příslušný program je popsán jako konečná posloupnost uspořádaných množin permutací. Množiny této posloupnosti jsou neprázdné a konečné. Vstup programu je sekvence čísel, která vybírá z každé množiny vždy jednu permutaci do posloupnosti, v níž je pořadí permutací určeno pořadím množin. Výstupem programu je pak permutace, která vznikne složením permutací této posloupnosti. [2, 3]*

Mějme permutační program, jež je posloupnost čtyř uspořádaných množin permutací (π_1, π_2), (π_3, π_4), (π_5) a (π_6, π_7, π_8). Jednotlivé permutace vypadají následovně:

První množina: $\pi_1 = (1, 4, 2, 3)$, $\pi_2 = (1, 3, 2, 4)$

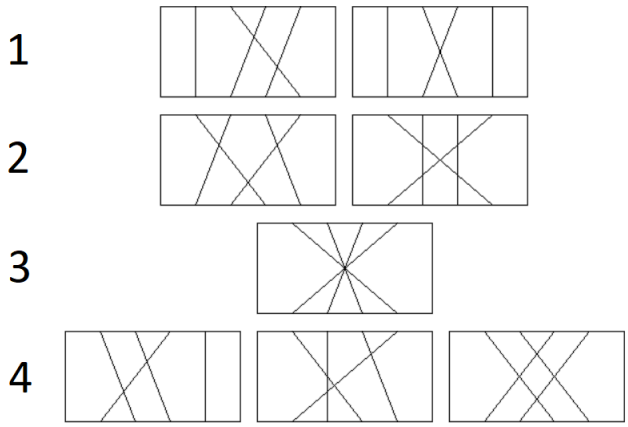
Druhá množina: $\pi_3 = (3, 1, 4, 2)$, $\pi_4 = (4, 2, 3, 1)$

Třetí množina: $\pi_5 = (4, 3, 2, 1)$

Čtvrtá množina: $\pi_6 = (2, 3, 1, 4)$, $\pi_7 = (3, 2, 4, 1)$, $\pi_8 = (3, 4, 1, 2)$

Graficky je tento permutační program zobrazen na Obrázku 3. Množinám permutací, které jsou součástí posloupnosti permutačního programu, budeme v rozsahu této práce říkat **vrstvy** permutačního programu. A **velikost vrstvy** bude určena počtem prvků této množiny. Na obrázku jsou vrstvy očíslovány. Vidíme, že jednotlivé vrstvy jsou zobrazeny pod sebou a permutace náležící do stejné vrstvy jsou zobrazeny vedle sebe. Vrstvy jsou uspořádány shora dolů tak, jak se vyskytují v posloupnosti permutačního programu a permutace jsou zobrazeny zleva doprava podle pořadí ve vrstvě. Např. první vrstva programu z obrázku by byla rovna (π_1, π_2), tedy jedná se o první uspořádanou množinu posloupnosti permutačního programu, a její velikost je 2. Vrstvám, které mají velikost 1, budeme říkat **statické vrstvy**.

Vstup permutačního programu je sekvence čísel, která vybírá z každé vrstvy vždy jednu permutaci do posloupnosti, v níž je pořadí permutací určeno pořadím vrstev, ze kterých jsou vybírány. Permutace ze statických vrstev jsou vybrány automaticky, protože máme na výběr



Obrázek 3: Grafické zobrazení permutačního programu

jenom jednu možnost. Vstup proto neobsahuje čísla, které by vybírali ze statických vrstev. Pozice čísla v sekvenci vstupu značí, ze které vrstvy volíme, když nepočítáme statické vrstvy, a hodnota čísla určuje, která z permutací má být vybrána. [2, 3] Vstupy permutačních programů budeme psát do hranatých závorek.

Výstupem permutačního programu pro daný vstup je permutace vzniklá postupným skládáním permutací sekvence vybrané vstupem. [2, 3]

Dejme tomu, že vstupem programu z obrázku bude sekvence $[1, 1, 3]$. První jednička značí, že vybíráme první permutaci z první vrstvy, druhá jednička vybírá první permutaci druhé vrstvy a trojka vybírá třetí permutaci čtvrté vrstvy. Jelikož třetí vrstva je statická, je její permutace vybrána automaticky a číslo pro výběr z ní není ve vstupu obsaženo. Vybrali jsme tedy sekvenci $(\pi_1, \pi_3, \pi_5, \pi_8)$. Výstupem programu pak bude $\pi_9 = \pi_8 \circ \pi_5 \circ \pi_3 \circ \pi_1 = (4, 1, 2, 3)$. Bude-li vstupem našeho permutačního programu sekvence $[2, 1, 2]$ výstupem programu bude $\pi_{10} = \pi_7 \circ \pi_5 \circ \pi_3 \circ \pi_2 = (2, 3, 1, 4)$.

Počet různých možných vstupů programu lze snadno vypočítat. Jenom navzájem vynásobíme velikosti jednotlivých vrstev programu. Pro program z obrázku tedy existuje $2 \cdot 2 \cdot 1 \cdot 3 = 12$ různých vstupů. Počet různých výstupů je však těžké určit. Pro různé vstupy totiž můžeme dostat stejné výstupy. Např. výstup $(4, 1, 2, 3)$ jsme dostali pro vstup $[1, 1, 3]$ a žádný jiným vstupem ho nezískáme, ale pro $[2, 1, 2]$ jsme dostali výstup $(2, 3, 1, 4)$ a stejný dostaneme i pro vstup $[2, 2, 1]$. Počet výstupu záleží především na podobě a počtu permutací náležících do jednotlivých vrstev. Maximální možný počet výstupů je $n!$, kde n je počet prvků permutací a další horní hranicí je počet možných vstupů, kterých může, ale nemusí být méně než $n!$. Když jich je více máme naopak jistotu, že aspoň pro nějaké různé vstupy dostaneme stejný výstup. Minimum různých výstupů je pak samozřejmě 1, pokud by všechny vstupy vraceli shodný výstup. To by nastalo v případě, kdyby každá vrstva obsahovala pouze vícekrát stejnou permutaci.

3 Popis aplikace a hry

3.1 Použitý programovací jazyk

Rozhodl jsem se, že aplikaci napíši v jazyce C# ve vývojovém prostředí **Visual Studio 2017**, které je pro vývoj C# aplikací uzpůsobeno. Co mě vedlo k využití právě programovacího jazyku C#? Jsem blíže obeznámen s jazyky C++, Java a C#. Se C# mám určitě nejvíce zkušeností. Je to první programovací jazyk, se kterým jsem byl kdy seznámen a to v době studia na střední škole. Aplikace má mít UI a nikdy jsem nedělal UI v C++ a v Javě jsem jich vytvořil jen pár. S knihovnami pro vytváření C# uživatelského rozhraní jsem naopak dobře seznámen a *Visual Studio* obsahuje nástroje pro jeho návrh. Navíc syntaxe C# má klíčové slova, které umožňují jednoduchou implementaci často používaných konstrukcí, tudíž pro programátora je psaní C# kódu pohodlnější, než třeba v Javě. Výpočetní rychlost sice není tak rychlá jako v C++, ale naše aplikace nepotřebuje počítat nic složitého. Trochu větší nevýhodou je nekompatibilita aplikací s platformami jinými, než je *Microsoft Windows*, ale pomocí *Mono frameworku* je možné aplikace překládat a spouštět i na jiných platformách.

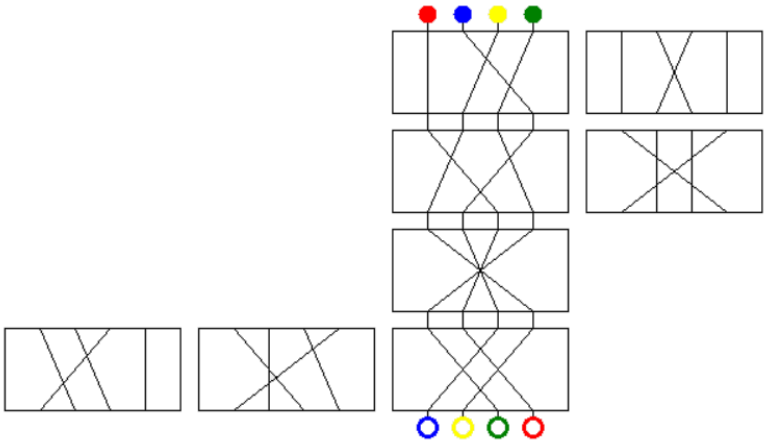
3.2 Struktura programu

Program je rozdělen do dvou projektů. Jedním je dynamická knihovna (.dll) a druhým spustitelná Windows Forms aplikace (.exe). Dynamická knihovna je nazvaná *PermutationPrograms* a obsahuje třídy, které reprezentují hry s permutační programy, a pomocnou třídu umožnující ukládání her. Důvodem implementace těchto tříd do samostatné knihovny je možnost, použít tyto třídy v budoucnu i v jiné aplikaci. *Game with permutation programs* je spustitelný soubor, který obsahuje třídy představující okna a ovládací prvky uživatelského rozhraní aplikace. Aplikace zabrání svému vícenásobnému spuštění na systému v jednom okamžiku. Je to z toho důvodu, že by si tyto aplikace mohli navzájem přepisovat uložená data. Neexistuje snad žádný důvod, proč by uživatel chtěl mít aplikaci spuštěnou vícekrát, tudíž by to nemělo být problémem.

3.3 Cíl a vzhled hry

Předtím než se začneme zabírat samotnou implementací, je potřeba objasnit, co je vlastně cílem hry, jak vypadá a jak bude fungovat. Jak bylo řečeno v Sekci 2.4, permutační program pro vstup vrací určitý výstup, který je výsledkem složení vstupem vybraných permutací. Úkolem hráče hry je nalézt vstup, pro který program vrací požadovaný výstup. Jak vypadá hra pro permutační program z Obrázku 3, je ukázáno na Obrázku 4.

Můžeme vidět, že hra využívá grafického zobrazení permutačních programů (Obrázek 3). Vstup permutačního programu je určen permutacemi, které jsou spojeny krátkými vertikálními čarami a jsou mezi barevnými kruhy a kružnicemi. Na obrázku je tedy navolen vstup $[1, 1, 3]$. Hráč může posouvat vrstvy doprava či doleva a tím mění vstup permutačního programu. Když je s navoleným vstupem spokojen, spustí animaci, při které se kruhy pohybují podél čar dolů



Obrázek 4: Vzhled hry

a správný vstup byl nalezen, pokud všechny kruhy skončí dole uvnitř kružnic se stejnou barvou. Pokud k tomu dojde je hráči oznámeno, že našel řešení hry. To jak kruhy při průchodu permutacemi mění pozice reprezentuje postupné skládání vybraných permutací a pořadí kružnic dole pak odpovídá výstupu, který chceme dostat. Hledaným výstupem hry z obrázku je permutace $(4, 1, 2, 3)$. Jak bylo řečeno v Sekci 2.4, tak tento výstup dostaneme pro právě navolený vstup $[1, 1, 3]$.

4 Implementace permutačních programů

V této části si projdeme implementační detaily tříd, které tvoří jádro naší aplikace, jímž jsou permutační programy. Na Obrázku 5 je třídní diagram pro třídy reprezentující permutační program. Základem je třída **Permutation** (Sekce 4.1) reprezentující permutaci. Z permutací se skládají jednotlivé vrstvy, v našem případě instance třídy **Layer** (Sekce 4.2). Třída **Game** (Sekce 4.3) představuje hru. Obsahuje samotný permutační program, který je složen z vrstev, a požadovaný výsledek hry. Nyní se podíváme na detaily jejich implementace.

4.1 Permutation

Vlastnosti

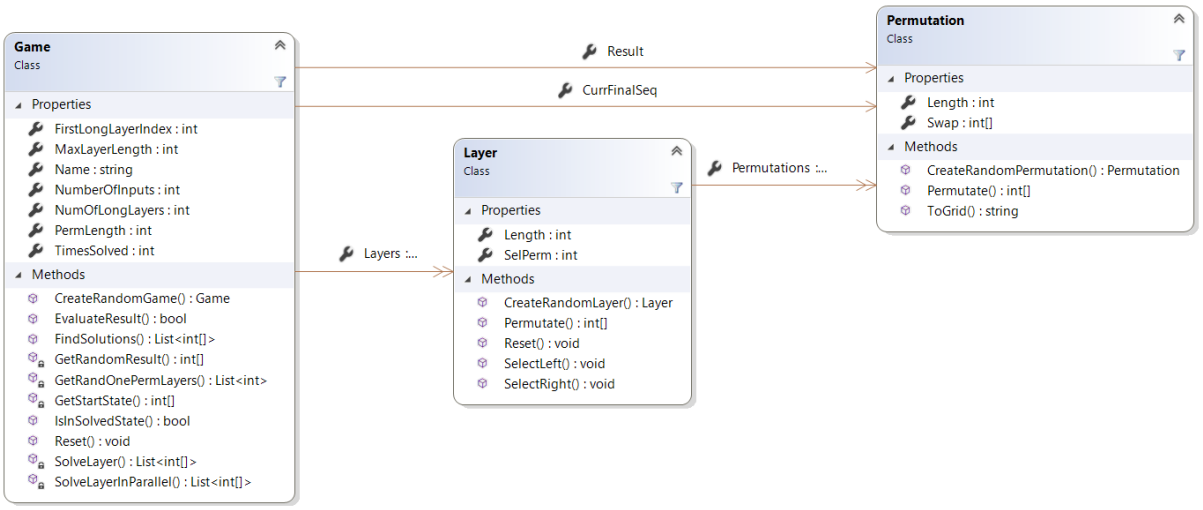
Permutation má dvě veřejné vlastnosti určené pouze ke čtení **Length** a **Swap**. *Swap* je samotná permutace reprezentovaná polem typu *int* a *Length* je počet prvků množiny, na kterou se permutace zobrazuje a vrací tedy hodnotu *Swap.Length*. Např. pro permutaci, kterou si v kódu pojmenujeme $myPerm = (3, 4, 2, 1, 5)$, bude $Swap = [3, 4, 2, 1, 5]$ a $Length = 5$. Dále má *Permutation* ještě indexer, který vrací hodnotu *Swap* na daném indexu. Pro permutaci *myPerm* by platilo $myPerm[3] = 1$ a je to tedy zkrácený zápis pro *myPerm.Swap*[3].

Metody

Konstruktory Třída má dva konstruktory. Jeden, jehož parametrem je pole pro *Swap*, a druhý, jehož parametry jsou *string* a hodnota očekávané velikosti pole *Swap*. Druhý konstruktor je v programu většinou použit, jelikož permutace převážně vytváří uživatel a ten je zadává v textové podobě. *String* pro permutaci *myPerm* by byl "3 4 2 1 5". Jedná se o jednořádkový zápis permutace bez čárek a závorek. Je samozřejmě nutné kontrolovat, zda *string* opravdu reprezentuje platnou permutaci. Za předpokladu, že *string* obsahuje čísla oddělené mezerami, konstruktor může vyhazovat následující výjimky:

- **NotEnoughValuesException**: *String* obsahuje méně čísel, než je hodnota druhého parametru.
- **WrongValueException**: *String* obsahuje neplatné číslo. Číslo musí být menší nebo rovno hodnotě druhého parametru a větší než 0.
- **MultiplicityOfValueException**: *String* obsahuje jedno číslo vícekrát. V permutaci musí být všechny prvky jedinečné.

CreateRandomPermutation(Random, int) Statická veřejná metoda, která vrací náhodně generovanou instanci *Permutation* s určeným počtem prvků. Pořadí prvků je generováno pomocí předané instance *Random*.



Obrázek 5: Třídy reprezentující permutační program

ToGrid() Jedná se o opak druhého konstruktoru. Vrací *string*, ze kterého by instance vznikla. Nevyužívá se pro tento účel metoda *ToString*, protože ta vrací standardní jednořádkový zápis.

Permutate(int[]) Toto je nejdůležitější metoda a její kód je na výpisu 1. Zobrazuje pole typu *int* předané jako parametr, na nové pole vzniklé permutováním pomocí vlastnosti *Swap*. Teoreticky by tato metoda mohla být generická, takže by dokázala permutovat pole jakéhokoliv typu a byla tudíž univerzálnější. Jelikož však v aplikaci používáme pouze pole typu *int*, tak nám bude stačit méně univerzální tvar. Ukážeme si, jak metoda funguje na příkladu. Mějme permutaci *myPerm*, jejíž *Swap* je roven $[3, 4, 2, 1, 5]$ a zavoláme na ni funkci *Permutate*, kde vstupním parametrem bude pole $[5, 2, 3, 4, 1]$. *myPerm* zobrazuje 1 na 3, což znamená, že zobrazuje 1. prvek prvek vstupu na 3. prvek výstupu. V našem případě se 1. prvek 5 posune na 3. pozici, 2. prvek 2 na 4. pozici atd. Celkovým výstupem pak bude pole $[4, 3, 5, 2, 1]$.

```
public int[] Permutate(int[] val)
{
    int[] newVals = new int[Length];
    for (int i = 0; i < Length; i++)
    {
        newVals[Swap[i] - 1] = val[i];
    }
    return newVals;
}
```

Výpis 1: Metoda Permutate třídy Permutation

4.2 Layer

Vlastnosti

Layer má tři vlastnosti. **Permutations** je list instancí typu *Permutation*, které jsou prvky vrstvy. **Size** je velikost vrstvy (*Permutations.Count*) a **SelPerm** určuje index permutace v listu *Permutations*, jež je vybrána do sekvence permutací, jejichž složením vznikne výstup perm. programu. Všechny tyto vlastnosti jsou veřejné pro čtení a pro *SelPerm* je umožněn privátní zápis. Tato třída má také indexer, který vrací permutaci na určitém indexu v *Permutations*. Jeho cílem je opět zjednodušení kódu, očividně *myLayer[0]* je kratší než *myLayer.Permutations[0]*.

Metody

Konstruktor *Layer* má pouze jeden konstruktor, jehož parametrem je *List<Permutation>*, který je přiřazen do vlastnosti *Permutations*.

CreateRandomLayer(Random, int, int) Statická veřejná metoda vracející novou náhodně generovanou instanci třídy *Layer* se specifickou velikostí a s permutacemi o určitém počtu prvků (hodnoty předány pomocí parametrů). Parametrem je předána reference na instanci třídy *Random*, kterou metoda používá pro generování náhodných permutací.

Reset() Jedná se o veřejnou metodu, která nastavuje vlastnosti vrstvy na výchozí hodnoty. V našem případě mění pouze *SelPerm*. Její výchozí hodnotou je index permutace, která se nachází uprostřed listu *Permutations*. Samozřejmě když je počet permutací sudý, potom máme dvě prostřední permutace. Pak vybereme tu s menším indexem. Ve všech případech dostaneme požadovaný index celočíselným dělením $(Size - 1)/2$. Tato metoda je volána také v konstruktoru.

SelectLeft(), SelectRight() Veřejné metody, jež manipulují s hodnotou vlastnosti *SelPerm*. *SelectLeft()* dekrementuje a *SelectRight()* inkrementuje *SelPerm* o jedna. Přitom kontrolují, aby byla hodnota vždy větší nebo rovna 0 a menší než *Size*.

Permutate(int[]) Veřejná metoda, jež volá metodu *Permutate* na vybranou permutaci vrstvy a vrací její výsledek.

4.3 Game

Vlastnosti

Game má větší počet vlastností, proto si je probereme podrobněji. Všechny vlastnosti třídy jsou veřejné a až na *TimesSolved* umožňují pouze čtení.

Name Vlastnost typu *string*. Jedná se o jméno hry, které usnadňuje její odlišení od ostatních her.

Layers List vrstev tvořící permutační program hry. Pořadí v listu určuje, jaké je pořadí vrstev v posloupnosti permutačního programu. Pro zjednodušení kódu má *Game* i indexer, který vrací *Layer* na daném indexu.

Result Permutace, která je hledaným výsledkem hry, tzn. má vzniknout složením vybraných permutací.

MaxLayerSize Vlastnost, jejíž hodnota je maximum z velikostí vrstev v *Layers*. Využívá se převážně při tvorbě uživatelského rozhraní.

NumOfLongLayers Počet vrstev větší velikosti než 1. Využívá se při hledání všech vstupů, které jsou řešeními hry.

PermLength Tato vlastnost obsahuje číslo, které je rovno hodnotě vlastnosti *Length* permutací, které tvoří vrstvy permutačního programu. Jejím smyslem je zjednodušení psaní kódu, protože bychom mohli tuto informaci získat z libovolné permutace. Např. mějme proměnnou typu *Game* pojmenovanou *myGame*. Pak bychom hodnotu získali pomocí *myGame.Layers[0].Permutations[0].Length* nebo použitím indexerů *myGame[0][0].Length*. Kód *myGame.PermLength* je stejně dlouhý jako ten s indexery a navíc více vypovídá o tom, co jím získáváme.

TimesSolved Vlastnost nesoucí informaci o tom kolikrát uživatel úspěšně vyřešil hru.

NumberOfInputs Vrací číslo rovné počtu možných vstupů permutačního programu.

FirstLongLayerIndex Vrací index první vrstvy, která má velikost větší než 1.

CurrFinalSeq Vrací instanci *Permutation*, jejíž *Swap* je roven sekvenci čísel, kterou bychom dostali, po permutování sekvence $(1, 2, \ldots, PermLength)$ momentálním vstupem permutačního programu.

Metody

Konstrutor Parametry konstruktoru jsou proměnné, které jsou dosazeny do *Layers*, *Result*, *Name* a *PermLength*. Jsou nalezeny hodnoty pro *MaxLayerSize* a *NumOfLongLayers*.

CreateRandomGame(Random, int , int, int) Statická veřejná metoda vracející náhodně generovanou hru na základě počtu vrstev, maximální velikosti vrstev a počtu prvků permutací předaných v parametrech. Je předána i reference na instanci třídy *Random*, kterou využíváme ke generování náhodných čísel.

GetRandOnePermLayers(Random, int) Statická privátní metoda, která je použita při generování náhodné hry v metodě *CreateRandomGame*. Číselným parametrem je celkový počet vrstev. Úkolem metody je vygenerovat náhodné indexy vrstev (vrácených v *List*u),

které budou velikosti 1. Protože tyto statické vrstvy podstatně snižují obtížnost hry, může jich být 0 až třetina celkového počtu vrstev a nižší počet je více pravděpodobný.

GetRandomResult(List<Layer>, int[], int) Statická privátní metoda vracející náhodný hledaný výsledek hry. Během generování vrstev v metodě *CreateRandomGame* se ke každé z nich generuje index permutace, jež bude součástí sekvence řešení. Tato metoda na základě struktury vrstev a těchto indexů zjistí hledaný výsledek hry.

GetStartState() Toto je privátní metoda vracející sekvenci čísel $(1, 2, \ldots, PermLength)$ jako pole typu *int*. Toto je očekávaná vstupní sekvence čísel, která je následně permutována jednotlivými vrstvami. Je použita např. při získávání hodnoty *CurrFinalSeq* a v dalších třídních metodách.

Reset() Veřejná metoda sloužící k resetování hry do výchozího stavu. V jejím těle se resetují všechny vrstvy.

EvaluateResult(int[]) Tato veřejná metoda vrací *bool* hodnotu podle toho, zda sekvence čísel předána jako parametr by byla výsledkem permutovaní sekvence získané metodou *GetStartState* permutací *Result*. Používá se tedy k vyhodnocení, zda uživatel nalezl řešení hry.

IsInSolvedState() Veřejná metoda, která volá ve svém těle metodu *EvaluateResult* s parametrem *CurrFinalSeq.Swap* a vrací její návratovou hodnotu. Zjistíme tedy, zda momentální vstup permutačního programu (určený vlastností *SelPerm* jednotlivých vrstev) je řešením hry.

SolveLayer(int, int[], int) Jedná se o rekurzivní metodu. Což znamená, že volá sama sebe ve svém těle, dokud není splněna nějaká podmínka, která tyto volání zastaví. Aby nedošlo k nekonečnému volání je nutné, aby se každým vnořeným voláním metody podmínka blížila ke svému splnění.

Smyslem této metody je najít všechny vstupy permutačního programu, které jsou řešením hry. V metodě je rekurze ukončena a kontroluje se, zda je druhý parametr *state* hledaný výsledek hry, jakmile je první parametr *layer* roven *Layers.Count*. Pokud však není podmínka splněna, permutujeme *state*, každou permutací vrstvy s indexem *layer* a získané sekvence používáme jako parametry pro další volání metod *SolveLayer* společně s prvním parametrem $layer + 1$. Tuto část metody můžeme vidět na výpise 2. Toto $layer + 1$ je právě ono přiblížení ke splnění podmínky, jež zaručuje ukončení vnořování. Do polí vracených v listu jako výsledky vnořených volání zapisujeme pozice permutací, jež jsou součástí řešení. Třetí parametr předáváme proto, abychom věděli, na kterou pozici v poli zapisovat index permutace. To protože nezapisujeme indexy permutací statických vrstev (byl by vždy 0) a je tedy odlišný od indexu vrstvy.

Všechny vstupy, které jsou řešením hry, dostaneme v listu polí typu *int* pomocí zavolání *SolveLayer(0, GetStartState(),-1)*. Z tohoto listu jsme schopni určit, zda je hra vůbec řešitelná a pokud ano, kolik je těchto řešení a jak vypadají.

```
for (int i = 0; i < Layers[layer].Size; i++)
{
    int[] tmp = Layers[layer][i].Permutate(state);
    List<int[]> retVal = SolveLayer(layer + 1, tmp, index);
    if (retVal.Count != 0)
    {
        if (Layers[layer].Size > 1)
            foreach (int[] res in retVal)
            {
                res[index] = i;
            }
        results.AddRange(retVal);
    }
}
```

Výpis 2: Rekurzivní volání v metodě SolveLayer

SolveLayerInParallel(int[]) Pro složitější permutační programy může nalezení všech řešení pomocí *SolveLayer* trvat dlouho. Počet volání metody *SolveLayer* by byl roven počtu možných vstupů programu +1. První zavolání metody je ono +1 a v ní je zavolána znovu pro každý možný vstup. Asymptotická časová složitost *SolveLayer* je rovna $O(k^n)$, což je opravdu hodně. Hodnota n je počet vrstev perm. programu a k je určeno velikostmi vrstev. Např. mějme perm. program se dvěma vrstvami o délce 3. Počet vstupů a tedy vnořených volání je $3 \cdot 3 = 3^2 = 9$. Přidáme-li vrstvu navíc opět o délce 3, pak by byl počet vstupů $3 \cdot 3 \cdot 3 = 3^3 = 27$.

Když se zamyslíme, tak nenajdeme efektivnější algoritmus, protože potřebujeme opravdu vyzkoušet všechny vstupy, zda nejsou řešeními hry. Ovšem pokud přidáním jedné vrstvy, časová složitost exponenciálně naroste, odebráním vrstvy by se měla exponenciálně snížit. A to právě simuluje tato privátní metoda. Nemůžeme samozřejmě doslova odebrat vrstvu z perm. programu, protože výsledný perm. program by nebyl stejný. Ale můžeme začít volat *SolveLayer* až od druhé nestatické vrstvy a to paralelně. Využijeme tedy vláken. V metodě permutujeme vstupní sekvenci (parametr), permutacemi vrstvy na indexu *FirstLongLayerIndex*. Získané výsledné sekvence používáme, jako vstupy pro metody *SolveLayer* s parametrem $layer = FirstLongLayerIndex+1$ a třetím parametrem rovným 0, které jsou volány ve vlastních vláknech. Použití vláken můžeme vidět na výpise 3.

Menší problém je, že vlákna nevracejí návratovou hodnotu. Dokážou však zapisovat do proměnných, a proto vlákno výsledek uloží do předem definovaného pole. Jednotlivá vlákna si také uložíme do pole, ať na ně pak můžeme zavolat metodu *Join()*, která donutí aplikaci čekat na dokončení vláken před dalším pokračováním. Po skončení všech vláken vracené listy sjednotíme do jednoho a vrátíme jako výsledek. Samozřejmě konečné zrychlení závisí na tom, kolik máme v počítači jader a kolik je jedno jádro schopno zpracovávat vláken v jednom okamžiku.

```
int layIndex = FirstLongLayerIndex;
List<int[]>[] retVals = new List<int[]>[Layers[layIndex].Size];
Thread[] threads = new Thread[Layers[layIndex].Size];
for (int i = 0; i<Layers[layIndex].Size; i++)
{
    int[] tmp = Layers[layIndex][i].Permutate(startState);
    int index = i;
    threads[i] = new Thread(
        () => { retVals[index] = SolveLayer(layIndex + 1, tmp, 0); });
    threads[i].Start();
}

for (int j = 0; j<threads.Length; j++)
{
    threads[j].Join();
    //...
```

Výpis 3: Použití vláken v metodě SolveLayerInParallel

FindSolutions() *SolveLayer* a *SolveLayerInParallel* jsou privátní metody a pro každou hru jsme schopni určit parametry, se kterými bychom je měli volat, abychom našli všechny řešení hry. Je však jasné, že bychom chtěli výsledky, těchto metod předat i ostatním třídám. Proto máme tuto metodu, která je veřejná a vrací všechny řešení hry tím, že volá jednu z nich ve svém těle. V případe *SolveLayerInParallel* musíme nejdřív permutovat počáteční stav všemi horními statickými vrstvami a až získanou sekvenci předat jako parametr, jak je vidět na výpise 4, protože metoda *SolveLayerInParallel* začíná pracovat až od vrstvy s indexem rovným *FirstLongLayerIndex*.

Problém je, že správa vláken systémem zpomaluje běh programu. Takže pokud by perm. program měl málo vrstev, tedy časová složitost *SolveLayer* by nebyla moc velká, tak čas strávený správou vláken by mohl být větší než úspora získaná souběžným během kódu. Změřil jsem tedy dobu potřebnou k nalezení všech řešení bez vláken a s vlákny pro různé permutační programy na svém notebooku s procesorem **AMD FX-7500 Radeon R7**.

```
public List<int[]> FindSolutions()
{
    int[] startState = GetStartState();

    if (NumberOfInputs < 65536)
        return SolveLayer(0, startState, -1);
    else
    {
        int layIndex = FirstLongLayerIndex;

        for (int i = 0; i < layIndex; i++)
            startState = Layers[layIndex].Permutate(startState);

        return SolveLayerInParallel(startState);
    }
}
```

Výpis 4: Metoda FindSolutions

Tento procesor má čtyři jádra, které neumožňují **multithreading**, a je tedy schopen současně pracovat s čtyřmi vlákny. Testované programy se lišili v počtu a velikosti vrstev, ale všechny měli $PermLength = 8$. K měření času jsem použil knihovní třídu *Stopwatch*. Výsledky měření jsou v tabulce 1, která je rozdělena na 3 části.

V první části je měněn počet vrstev a můžeme vidět, že správa jednoho vlákna zabírá asi 16 ms a při menším počtu vrstev se metoda vykoná v mnohém menším intervalu bez vláken. Ale jakmile je vrstev více než 8, tak pro vrstvy o velikosti 5 se vlákna už vyplatí. V druhé části tabulky vidíme závislost na velikosti vrstev. Z naměřených údajů můžeme vydedukovat, že vlákna se vyplatí pro 10 vrstev velikosti větší než 2, pro 9 vrstev velikosti větší než 3 a pro 8 vrstev velikosti větší než 4. Pro programy s velikostí vrstev 2 musíme mít až 16 vrstev, aby se vlákna aspoň vyrovnali času běhu aplikace bez nich. Poslední věc, kterou testujeme a výsledky jsou zaneseny ve třetí části tabulky, je to zda je důležitý počet vláken. Jelikož tvoříme tolik vláken, kolik je velikost první větší vrstvy, tak jsem testoval programy s různou velikostí první vrstvy. S výsledků je vidno, že pokud byli vlákna efektivnější pro daný počet vrstev dané velikosti budou rychlejší, ať jich je libovolné množství.

Na základě těchto měření můžeme použít vlákna vždycky. Vlákna jsou pomalejší v řádu desítek milisekund, čehož si uživatel ani nevšimne. Naopak jsou ale rychlejší až v řádu vteřin, což už každý zaznamená. Jelikož ale chceme dostat z naší aplikace maximum, tak vlákna použijeme pro perm. programy s aspoň $2^{16} = 65536$ vstupy. 2^{16} protože to je počet vstupů pro 16 vrstev velikosti dva, pro které jsme dostali téměř stejné časové intervaly. Podobné výsledky máme i pro 8 vrstev velikosti 4 a opravdu $4^8 = 65536$.

Počet vrstev	Velikost vrstev (první/zbytek)	Čas bez vláken [ms]	Čas s vlákny [ms]
1	5	1	84
2	5	1	82
3	5	1	78
4	5	1	85
5	5	2	82
6	5	10	80
7	5	46	100
8	5	240	175
9	5	1162	660
10	5	5708	2994
8	2	1	26
8	3	6	34
8	4	44	60
9	2	1	30
9	3	15	66
9	4	204	127
10	2	2	32
10	3	39	60
10	4	636	405
16	2	75	79
7	2/5	16	44
7	4/5	34	100
8	2/5	84	79
8	4/5	186	131
9	2/5	424	296
9	4/5	841	442
10	2/5	2150	1472
10	4/5	4445	2113

Tabulka 1: Doba potřebná k nalezení všech řešení hry bez vláken a s vlákny

4.4 FileManager

Úkolem této pomocné třídy je ukládání a nahrávání instancí, jež budeme chtít zachovat i po zavření aplikace. Jedná se o statickou třídu, která má pouze dvě funkce **Save(object, string)** a **Load(string)**. Implementace je primitivní, využíváme *BinaryFormatter*. V metodě *Load* je-

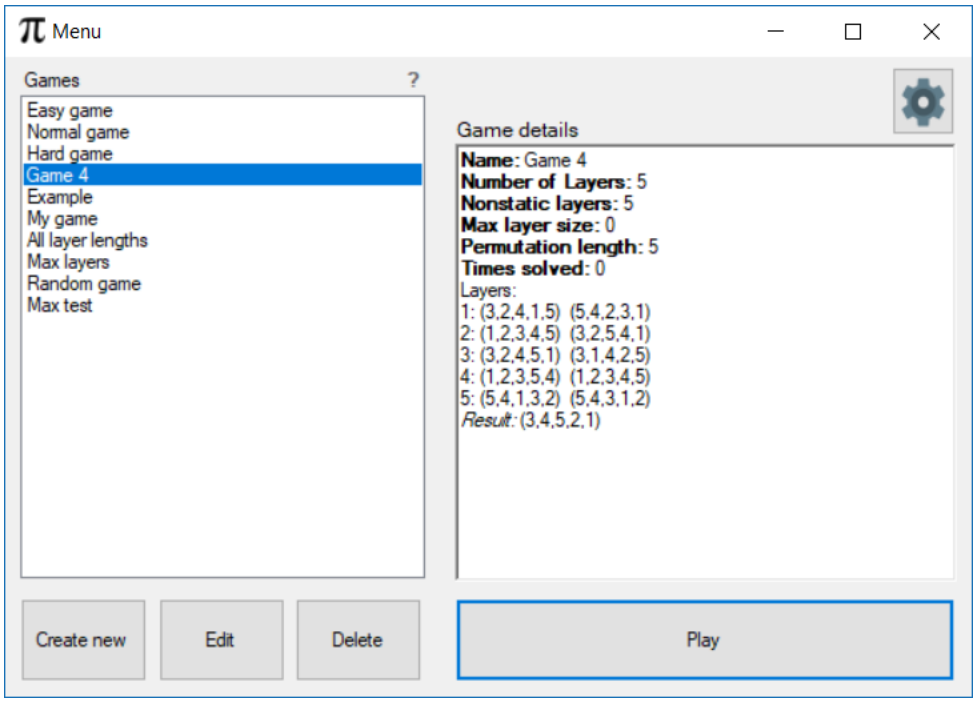
nom kontrolujeme, zda existuje soubor, ze kterého chceme číst, jež určuje druhý parametr, a pokud ne, metoda vrací *null* místo deserializovaného *object*u. Pokud je vrácen *object*, pak je ho potřeba přetypovat na požadovaný typ. Ovšem při čtení dat může dojít k vyhození výjimek, kvůli změně struktury dat. V metodě *Load* proto máme ještě *try-catch* blok, kde odchytáváme případné výjimky a metoda pak vrací *null*. Pravděpodobně bychom stejně nechtěli instance, které mají jinou strukturu, uchovávat v našem programu. My budeme ukládat vytvořené hry a já jednoduše počítám s tím, že nikdo nebude výrazně zasahovat do struktury tříd, které je tvoří. Stále je možné bez obav do nich psát nové metody a přidávat vlastnosti určené pro čtení, které vrací hodnoty na základě už uložených proměnných. Nové vlastnosti, které uchovávají data, však budou obsahovat *default* hodnoty, což může způsobit problémy, především pokud by šlo o referenční typy.

5 Uživatelské rozhraní

V této kapitole si projdeme návrh UI z pohledu uživatele. Popíšeme si jeho funkce a vzhled. Celé UI je psáno v angličtině, vzhledem k tomu že je to celosvětově rozšířený jazyk a většina lidí mu rozumí. Tím je zaručeno, že naší aplikaci bude rozumět co největší počet lidí. Snažil jsem se, aby bylo UI co nejvíce intuitivní, přehledné a pro uživatele pohodlné používat.

5.1 Menu

Toto okno se otevře po spuštění aplikace. Obsahuje list vytvořených her a funguje jako rozcestník, který umožňuje otvírat ostatní okna. Jeho vzhled je na Obrázku 6. Můžeme vidět seznam her, 1 textové pole a 5 tlačítek. Okno, jako většinu oken, lze zvětšovat. Stisknutím tlačítka *Escape* na klávesnici se okno zavře.



Obrázek 6: Menu

5.1.1 Seznam her a textové pole

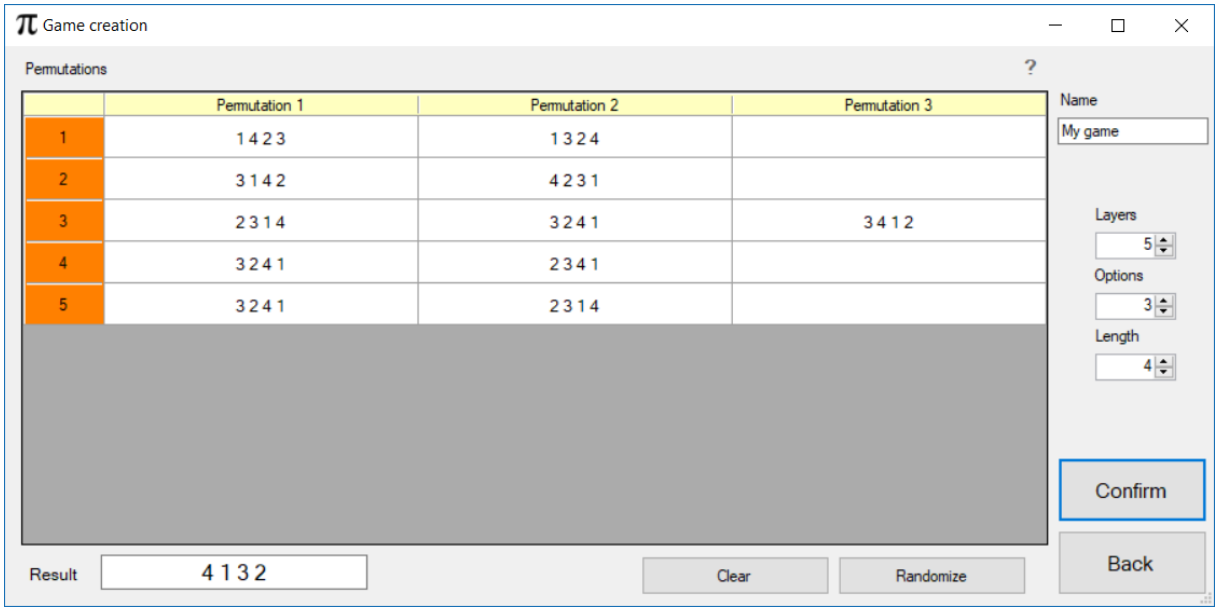
Jednotlivé hry ze seznamu her **Games** lze kliknutím vybírat (na obr. 6 je vybrána *Game 4*) a textové pole **Detail** zobrazuje detailní informace o aktuálně vybrané hře. Na základě těchto informací si uživatel může vybrat pro něho tu správnou hru. Textové pole zvětšuje svoji velikost společně s velikostí okna, pokud by se text do něj nevlezl. Také když uživatel zvětšuje okno, tak pole zvětšuje svoji velikost společně s velikostí písma. Pořadí her v seznamu lze měnit. Pokud klikneme na hru, podržíme tlačítko myši a zároveň stiskneme klávesu Shift, můžeme vybranou

hru přetáhnout na novou pozici, která je zvolena uvolněním tlačítka myši. Jelikož pořadí her lze měnit pouze tímto způsobem, na což by pravděpodobně uživatel sám nepřišel, je nad seznamem her malý otazník. Pokud na něj najedeme myší, zobrazí se nápověda, která informuje o této možnosti. Těchto otazníků je v aplikaci více a vždycky informují uživatele o funkcích konkrétních kláves.

5.1.2 Tlačítka

V okně je 5 tlačítek. Tlačítka **Create new**, **Edit** a **Delete** manipulují s hrami. Tlačítko *Delete* jednoduše vybranou hru odstraní ze seznamu, kdežto *Create new* a *Edit* otevírají okno (Sekce 5.2) pro tvorbu a editaci her. Tlačítko **Play** pak otevře okno, ve kterém lze vybranou hru hrát (Sekce 5.4). Při otvírání oken pro tvorbu a hraní her se toto okno schová a při jejich zavření, se znovu objeví. Tlačítko s ozubeným kolečkem otevře okno s nastavením hry (Sekce 5.5).

5.2 Okno tvorby her



Obrázek 7: Okno tvorby hry

Toto okno (Obr. 7) umožňuje vytváření nových a editaci už existujících her. Je otevřen po stisknutí tlačítek *Create new* nebo *Edit* okna *Menu* (Sekce 5.1). Kromě křížku ho lze uzavřít stisknutím tlačítka **Back** či klávesou *Escape*. Ve všech těchto případech jsou veškeré provedené změny zrušeny. Okno podporuje změnu velikosti a mění přitom velikost tabulky.

5.2.1 Parametry hry

V okně máme tři textová pole, které umožňují zadávat pouze čísla. Vlevo od těchto textových polí jsou ještě dvě malé tlačítka se šipkami, kterými lze nastavovat zadanou hodnotu. Zadané hodnoty definují parametry hry. **Layers** vyjadřuje počet vrstev našeho perm. programu, **Options** maximální velikost vrstvy a **Length** počet prvků permutací. Aby byli hry rozumně složité a dali se přehledně zobrazit na obrazovku, mají následující minima a maxima. Vrstev může být 1 až 10 velikosti 2 až 5. Počet prvků permutací je minimálně 2, protože jenom jeden prvek nemůže měnit pořadí, a maximální počet je 8, aby výsledná hra byla ještě přehledná. Textové pole neumožní zadání jiných hodnot.

5.2.2 Tabulka

Tabulka určuje strukturu permutačnímu programu. Do jednotlivých buněk zadáváme permutace jednořádkovým zápisem bez závorek a jednotlivé čísla jsou odděleny mezerami místo čárek. Číslo řádku buňky určuje vrstvu a číslo sloupce pořadí permutací. Tabulka reaguje na změny parametrů hry. Počet zobrazených řádků odpovídá hodnotě *Layers*, počet sloupců hodnotě *Options* a požadovaný počet čísel v buňkách hodnotě *Length*. Při snížení hodnoty *Layers* se řádky schovají a při zvýšení zobrazí i s předtím zadanými permutacemi. To samé se děje se sloupci při změně hodnoty *Options*. Tabulka se zvětšuje při zvětšování okna a přitom se automaticky zvětšuje i velikost písma buněk.

Jelikož text popisující permutace obsahuje pouze čísla od jedné do hodnoty *Length* tak kromě těchto čísel a **Backspace**, jsou všechny ostatní klávesy ignorovány. Mezery mezi čísla se vkládají automaticky. Buňka nedovolí zapsat i čísla, které už obsahuje, a tím zabraňuje vytvoření neplatné permutace. *Backspace* standardně maže text. Je taky možné označit i část textu a tu pak smazat či přepsat pod podmínkou, že nevznikne chybná permutace. Ovšem stále se může stát, že buňky budou obsahovat chybné permutace, pokud už máme nějaké permutace vyplněny a následně snížíme hodnotu *Length*. Pak můžou buňky obsahovat moc velká čísla. Proto když dojde ke změně *Length*, je text buněk obsahující chybnou permutaci obarven na červeno, aby byl uživatel na tuto okolnost upozorněn. Po opravení permutace je text obarven zpátky na černo. Tabulka umožňuje kopírování a mazání obsahu buněk, řádků a sloupců pomocí klávesových zkratek **Ctrl + C**, **Ctrl + V** a **Ctrl + X**, které mají svoje standardní funkce, jak je známe z textových editorů. Mazat obsah buněk lze i tlačítkem **Delete**, ale pozor pokud je označen řádek, tak celý řádek bude odstraněn i s hlavičkou. Pokud chceme pouze vymazat obsah buněk a řádek ponechat, tak použijeme *Ctrl + X*. Na možnost použití těchto kláves je uživatel informován najetím kurzoru myši na otazník nad pravým horním rohem tabulky. Buňku, jejíž obsah chceme kopírovat či vymazat, stačí vybrat kliknutím a pak už jenom stiskneme příslušné klávesy. Buňky řádků a sloupců se vybírají kliknutím na jejich hlavičkové buňky. Při mazání a kopírování řádků a sloupců jsou brány v potaz pouze zobrazené buňky.

5.2.3 Textové pole

Součástí okna jsou dvě textová pole. Obsahem textového pole **Name** je jméno hry. Do pole **Result** zadáváme výstup perm. programu, jehož nalezení je cílem hry. Pole *Result* má stejné chování jako buňky tabulky, tzn. kontroluje klávesové vstupy a obarvuje text pokud vznikla neplatná permutace.

5.2.4 Tlačítka

Okno má pět tlačítek. Tlačítko **Back** jednoduše zavře okno bez toho, aniž by byly změny uloženy. Tlačítko **Clear** vymaže text ze všech zobrazených buněk.

Vytváření hry může být zdlouhavé a složité, přitom uživatel může chtít rychle novou hru, kterou může vyzkoušet. Tlačítko **Randomize** generuje náhodnou hru na základě hodnot *Length*, *Options* a *Layers*, naplní buňky jejími permutacemi a vygeneruje i náhodný výsledek. Tyto náhodně generované hry mají výhodu. Permutace bývají výrazně odlišné a následkem toho je, že hra má malý počet řešení a není ji tedy jednoduché vyřešit.

Jak bylo řečeno v Sekci *Tabulka* může se stát, že po změně hodnoty *Length* dostaneme buňky s chybnými vstupy. Pokud by takovýchto buněk bylo velké množství, bylo by zdlouhavé a nepříjemné ručně je opravovat. Tlačítko **Fix inputs**, které není na Obrázku 7 vidět, automaticky opraví všechny zobrazené buňky odstraněním chybných čísel. Tlačítko se vyskytuje pod tabulkou mezi tlačítkem *Clear* a polem *Result* a je viditelné jenom pokud některé zobrazené buňky obsahují chybné vstupy. Kromě buněk tabulky opravuje i pole *Result*.

Po stisknutí tlačítka **Confirm** se okno snaží vytvořit novou hru ze zadaných dat. A následně zobrazí okno s informacemi (Sekce 5.3). Při vytváření se berou v potaz pouze zobrazené buňky a není nutné, aby vyplněné buňky byly na začátku řádku.

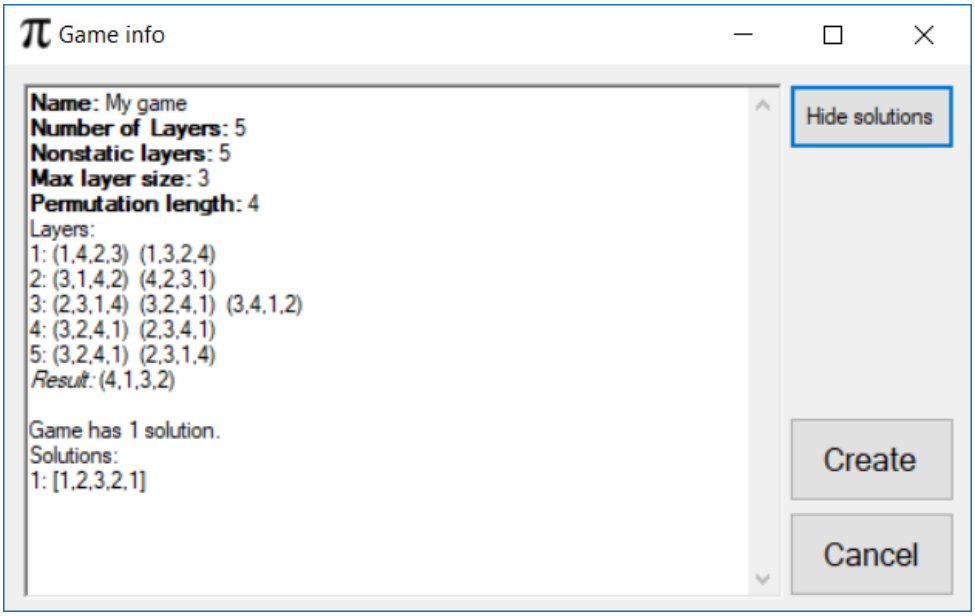
5.3 Informační okno

Informační okno po stisknutí tlačítka *Confirm* (Sekce 5.2.4) zobrazí informace o hře zadané v okně tvorby hry. Vypisuje buď chyby, které znemožnily vytvoření, nebo informace o nové hře, jak je tomu na Obrázku 8. Pokud se uživatel pokusí okno zavřít, pouze se schová a zobrazí se při novém stisku tlačítka *Confirm*. I když je okno zobrazeno může uživatel dál editovat údaje v okně tvorby hry, aby mohl na základě informací opravit chyby či změnit strukturu hry, pokud by se mu nelíbila. Provedené změny se ale projeví v informačním okně až po opětovném stisknutí tlačítka *Confirm*. Při zvětšování mění okno velikost textového pole společně s velikostí jeho textu.

5.3.1 Zobrazené informace

Hra nemusela být vytvořena z důvodu špatně zadaných dat. Chyby můžou být následující:

- Málo zadaných čísel v permutaci
- Příliš velká čísla v permutaci



Obrázek 8: Informační okno

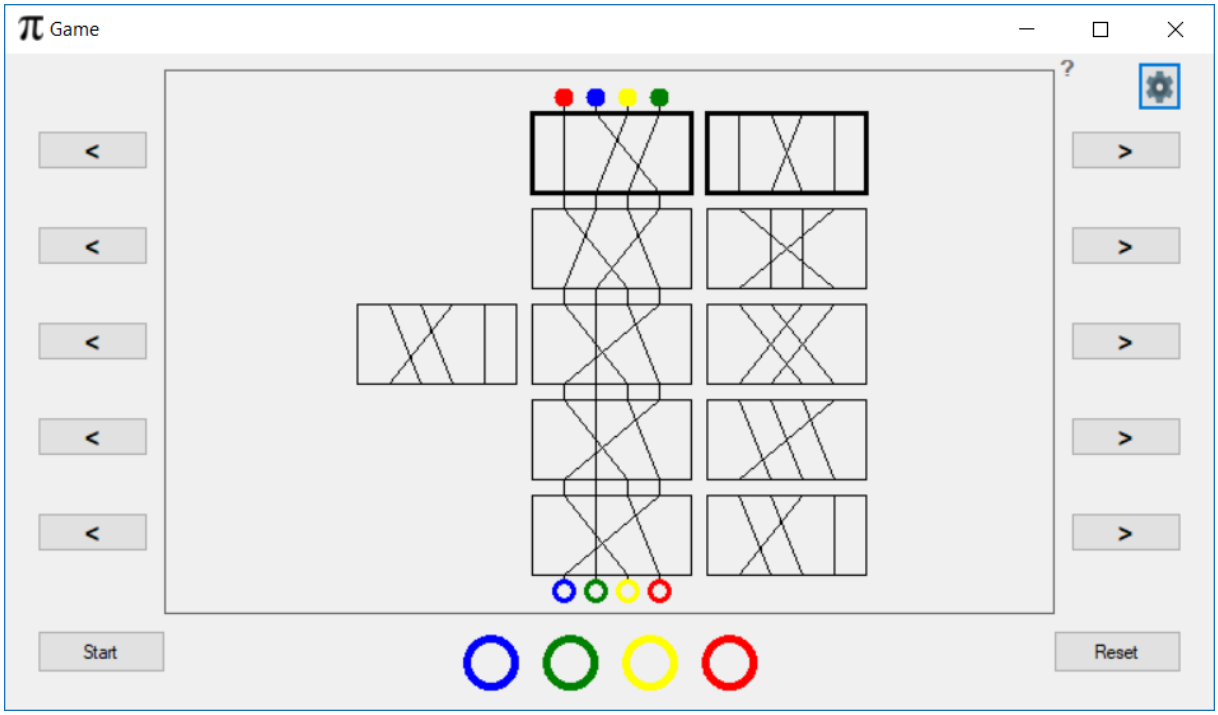
- Řádek neobsahuje ani jednu permutaci
- Všechny řádky obsahují jenom jednu permutaci
- Není vyplněno jméno hry
- Není vyplněno pole *Result*
- Hra nemá řešení

Pokud je to možné jsou u chyb uvedeny buňky či řádky, jež je způsobily.

Pokud se hru podařilo vytvořit, jsou její informace zobrazeny uživateli společně s počtem různých vstupů, jež jsou řešením. Jednotlivá řešení je možné si nechat zobrazit.

5.3.2 Tlačítka

Tlačítko **Cancel** schovává okno. Tlačítka **Create** a **Show solutions** jsou viditelné pouze v případě, že okno zobrazuje data o úspěšně vytvořené hře. *Show solutions* zobrazuje a schovává nalezená řešení hry, což jsou vstupy, kterými dostaneme požadovaný výstup. Řešení jsou nejdřív skrytá, aby uživatel nepřišel o možnost sám je najít, a je vypsán pouze jejich počet. Při stisknutí tohoto tlačítka se zobrazí seznam řešení a zároveň se změní text tlačítka na *Hide solutions*, jak je vidět na Obrázku 8. Řešení lze opakovaně schovávat a zobrazovat. Pokud je uživatel se hrou spokojen tlačítkem *Create* potvrdí vytvoření této hry. Po jeho stisknutí je okno zavřeno společně s oknem tvorby a uživatel je navrácen do *Menu*. Je nutno pamatovat, že hra bude odpovídat hře zobrazené v informačním okně. Změny provedené v okně tvorby po stisknutí tlačítka *Confirm* jsou ztraceny.



Obrázek 9: Okno s hrou

5.4 Okno hry

Pomocí okna hry hrajeme samotnou hru. Okno obsahuje hru, tlačítka pro nastavování vstupu (tlačítka se šipkami), tlačítko *nastavení*, tlačítko **Start** a tlačítko **Reset**. Okno umožňuje změnu velikosti, při čemž se mění rozměry hry a tlačítek se šipkami.

5.4.1 Hra

Hra funguje a vypadá stejně, jak je popsáno v Sekci 3.3. Hra na obrázku odpovídá té, která je tvořena na Obrázcích 7 a 8 a na obrázku je navolen vstup $(1, 1, 2, 1, 1)$. Hra je ohraničena černou čarou.

Druhou částí hry jsou čtyři větší kružnice u dolního okraje okna. Zobrazují výsledek posledního vstupu a uživatel může na jejich základě vstup opravit. Zobrazovaný výsledek je změněn, pouze pokud animace hry proběhla až do konce.

5.4.2 Tlačítka

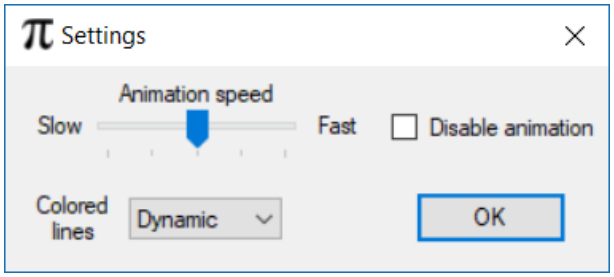
Tlačítko *Start* spouští a zastavuje animaci hry. Pokud animace běží mění svůj text na *Stop*. Tlačítko *Reset* vrací hru do stavu v jakém byla při otevření okna. Tlačítko s ozubeným kolečkem otevírá okno s nastavením hry (Sekce 5.5). Tlačítka se šipkami se vyskytují vedle nestatických

vrstev. Jejich smyslem je měnit vstup permutačního programu. Tyto tlačítka taky mění svoji výšku a velikost fontu na základě výšky okna.

5.4.3 Vstupy klávesnice

I když lze celou hru ovládat klikáním na tlačítka, tak to není zrovna pohodlné. Proto je možné ovládat hru i pomocí klávesnice. Na Obrázku 9 je vidět, že obdélníky první vrstvy jsou nakresleny tlustší čárou. To je momentálně vybraná vrstva a stisknutím **levé či pravé šipky** můžeme měnit její vybranou permutaci. Vybranou vrstvu můžeme měnit pomocí **šipek nahoru a dolů**. Tímto intuitivním ovládáním pomocí šipek tedy můžeme jednoduše měnit vstup permutačního programu. Klávesa **Enter** pouští a zastavuje animaci, takže má stejnou funkcionalitu jako tlačítko *Start*. Klávesa **Escape** pak standardně zavírá okno. O možnostech ovládání hry pomocí kláves se uživatel dozví najetím kurzoru myši na otazník, jež je u pravého horního rohu hry.

5.5 Nastavení hry



Obrázek 10: Nastavení hry

Pomocí tohoto okna můžeme měnit nastavení hry. Upravujeme vlastně nastavení animace. Toto okno je zobrazeno pomocí tlačítek s ozubeným kolečkem. Okno zůstává otevřeno dokud ho uživatel nezavře. Zůstává tedy viditelné i během přecházení mezi jednotlivými okny. Křížek, tlačítko **OK** nebo klávesa **Escape** schová okno.

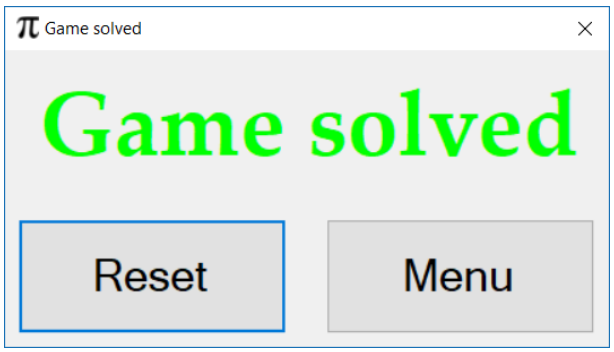
5.5.1 Možnosti nastavení

Animace může mít různou rychlost. Je zřejmé, že někdo může chtít podrobně sledovat, jak se permutace skládají a někdo jiný chce rychle vidět výsledek. Rychlost animace lze nastavit pomocí posuvníku **Animation speed**. Pokud by někoho zajímal jenom výsledek a animaci nechce vůbec spouštět pomocí políčka **Disable animation** ji lze úplně vypnout. Poslední možností je nechat čáry, po kterých se kruhy pohybují, obarvovat. Pro pole **Colored lines** máme tři možnosti: **None**, **Dynamic** a **Full**. V případě *None* zůstávají čáry neobarveny a naopak pro možnost *Full* jsou všechny čáry, které by následoval určitý kruh, zbarveny jeho barvou. Pomocí možnosti *Full* může uživatel vidět, po kterých čarách se bude kruh pohybovat a vidí, na které pozici nakonec skončí. Tato možnost docela zjednodušuje řešení hry, protože hned je možno poznat,

zda je navolený vstup správný či ne. V případě složitějších her může být opravdu těžké řešení najít a takto je možné si to zjednodušit. Možnost *Dynamic* je zde spíše pro zajímavější animaci. Tato možnost obarvuje čáry společně s pohybem kruhů. Vypadá to, jakoby kruhy postupně obarvovaly čáry, po kterých se pohybují.

5.6 Řešení nalezeno

Toto je nejjednodušší okno naší aplikace. Je zobrazeno v případě, že uživatel našel řešení. Pomocí tlačítek se pak můžeme buď vrátit do menu nebo resetovat hru. Pokud uzavřeme okno křížkem jsme navráceni do menu.



Obrázek 11: Hra byla vyřešena

6 Implementace UI

V této kapitole se podíváme na některé zajímavé detaily implementace UI. K implementaci využívám tříd z jmenných prostorů **System.Windows.Forms** obsahující třídy pro vytváření aplikací s uživatelským rozhraním, jež jsou založeny na systému Windows a **System.Drawing** poskytující přístup k základní grafickým funkcím rozhraní **GDI+**. Windows GDI+ je API, které umožňuje aplikacím používat grafiku a formátovaný text na obrazovce. [4] Více informací o třídách těchto jmenných prostorů lze nalézt v [5, 6].

6.1 FormMenu

Třída reprezentující okno *Menu* (Sekce 5.1). V konstruktoru se načítají pomocí *FileManager*u (Sekce 4.4) předtím všechny vytvořené hry (Sekce 4.3) a herní nastavení (Sekce 6.5.1). Okno ukládá hry pokaždé, když dojde k jejich změně (vytvoření, editace, smazání), aby v případě, že by došlo k selhání systému, nebyly změny ztraceny. Nastavení je uloženo pouze při zavření aplikace, jelikož jeho ztráta není velikým problémem, protože je otázkou pár sekund nastavit hru znovu. Okno obsahuje *ListBox*, *RichTextBox* a tlačítka.

6.1.1 ListBox

ListBox obsahuje seznam tzv. **item**ů, ze kterých je možno vybírat. Každý řádek reprezentuje jeden item a text řádků je definován metodou *ToString*, kterou dědí všechny typy z třídy *Object*. Itemy *ListBox*u jsou v našem případě vytvořené hry a jejich metoda *ToString* vrací hodnotu vlastnosti *Name*. Důležitými vlastnostmi *ListBox*u jsou **SelectedItem**, **SelectedIndex** a **DataSource**. *SelectedItem* je object odpovídající právě vybranému itemu a *SelectedIndex* je jeho index v seznamu. Pomocí *DataSource* můžeme nastavovat seznam objektů, které má *ListBox* obsahovat. Okno si proto udržuje ve vlastní proměnné list her. Zde je nutno podotknout, že změny provedené v tomto listu se nezobrazí v *ListBoxu*. Aby k tomu došlo je nutné nastavit *DataSource* na *null* a následně opětovně na náš list her.

6.1.2 TextBoxDetail

Tento *TextBox* obsahuje detaily právě vybrané hry. Jedná se o **RichTextBox**, který na rozdíl od normálního *TextBox*u umožňuje formátovat text. V našem případě jsme použili tlusté písmo a italiku. Text se mění při vyvolání události **SelectedIndexChanged** *ListBox*u. Nejedná se však ani o obyčejný *RichTextBox*, ale o **GameRichTextBox** (Sekce 6.1.4), který automaticky mění velikost textu při změně velikosti *RichTextBox*u. Pro vytvoření a formátování textu obsahujícího informace o hře jsem si vytvořil speciální třídu **GameRichTextFormatter** (Sekce 6.1.3).

6.1.3 GameRichTextFormatter

Tato třída je statická a její metody jako parametry přijímají hru, pro kterou chceme vypsat detaily, a *RichTextBox*, do kterého chceme psát. V jejích metodách tvoříme text pomocí *StringBuilder*u a formátujeme jej za použití metod a vlastností *RichTextBox*u.

6.1.4 GameRichTextBox

GameRichTextBox je odvozený z třídy *RichTextBox* a má navíc funkce, které při změně jeho velikosti zajišťují automatickou změnu velikosti textu. Zmíním tři zajímavé metody.

První je metoda *InitInstance(bool,int)*. Tato metoda sice není zas tak zajímavá funkcí, jelikož jejím smyslem je pouze inicializace proměnných, které jsou potřeba pro budoucí vypočítávání velikosti textu, ale zajímavý je důvod její existence. Inicializaci proměnných bychom přece mohli provést v konstruktoru, ale v tomto případě to tak není. V době volání konstruktoru totiž ještě nejsou nastaveny proměnné, jejichž hodnoty potřebujeme uložit. Tyto proměnné jsou rozměry textového pole, okna a textu. Tato funkce tedy potom, co jsou už všechny prvky nastaveny na očekávané hodnoty, zanese tyto údaje do instančních proměnných a tím si *TextBox* zapamatuje poměr velikosti písma k jeho velikosti a snaží se tento poměr udržovat. Musíme tedy pamatovat na její zavolání, protože jinak textové pole nebude nic dělat, a ideální je provést to ihned po inicializaci ovládacích prvků.

Druhá důležitá metoda je *UpdateTextBox(Game)*, která mění hru, pro niž vypisuje pole informace. V případě nutnosti zvětšuje pole i okno, pokud by se do něj nový text nevešel.

Poslední je metoda *ResizeText()*, která mění velikost textu a je zavolána při vyvoláni události *Resize*. Funguje to jednoduše. Podělíme nové rozměry se základními rozměry (uložené pomocí *InitInstance*) a dostanu poměr výšek a šířek. Vezmu z nich menší poměr a tím vynásobím základní velikost textu a dostanu velikost novou. Má to pár nedostatků, ale ty jsou zapříčiněny implementaci ovládacích prvků a ne tímto algoritmem. První problém je, že počítání s velikostmi není úplně přesné. Velikost písma je uložena jako *float* a velikost prvků jako *int*y. Tzn. že při počítání dochází k nepřesnostem kvůli zaokrouhlování hodnot. Abych omezil odchylky od očekávaných hodnot co nejvíce, tak *int* převádím před počítáním na *float*. Taky proto odvozuji novou velikost písma od základní velikosti ne od velikosti, která vznikla předchozí změnou, protože bych počítal už s jednou zaokrouhlenou hodnotou, kdežto ty základní jsou přesné a v *InitInstance* jsou už uloženy jako *float*. Druhou nevýhodou je, že ani *RichTextBox*y nejsou schopny zobrazovat libovolné velikosti písma. Správně by se měl text zvětšovat plynule, tak jak se zvětšuje *RichTextBox*, ale místo toho se zvětšuje skokově podle toho, která z hodnot zobrazitelné velikosti textu je blíže té nastavené. Kvůli tomu není poměr velikost textu ku velikosti textového pole vždy přesně zachován. A posledním problémem je, že části textu mohou občas problikávat, když dochází ke změně velikosti, což je zapříčiněno tím, že ovládací prvky se automaticky překreslují při změně velikosti a *RichTextBox*y neumožňují **DoubleBuffering** oproti třeba oknům či pa-

nelům. I přes tyto nedostatky se ale domnívám, že je to pro uživatele příjemnější, než kdyby se text nezvětšoval vůbec.

6.2 CreateGameForm

Tato třída reprezentuje okno tvorby her (Sekce 5.2), které umožňuje vytváření nových a editaci už existujících her. Je otvíráno jako dialogové okno, abychom rozeznali, zda uživatel potvrdil vytvoření hry. Má dva konstruktory jeden bez parametrů, ve kterém se nastavují výchozí hodnoty proměnných a je použit pro vytváření nové hry, a druhý s parametrem typu *Game* používaný pro editaci her už vytvořených, který volá konstruktor bez parametrů a plní ovládací prvky daty editované hry. Okno si také vytvoří instanci okna *InfoForm* (Sekce 6.3).

6.2.1 NumericUpAndDown

NumericUpAndDown je knihovní třída a jedná se ovládací prvek, reprezentující textové pole, do kterého je možno zadávat pouze číselné hodnoty. Lze nastavit minimální a maximální povolenou hodnotu a počet desetinných míst. Navíc jsou vedle textového pole dvě tlačítka, kterými lze číslo inkrementovat a dekrementovat o námi určenou hodnotu. Ve okně máme tři tyto prvky blíže popsány v Sekci 5.2.

6.2.2 Tabulka

Tabulka určuje strukturu permutačnímu programu. Jedná se o knihovní třídu **DataGridView**, která zobrazuje data pomocí tabulky ovládacích prvků. V našem případě je to tabulka *TextBox*ů. Tabulka reaguje na události *ValueChanged* prvků *NumericUpAndDown* schováváním a zobrazováním sloupců a řádků. Trochu problém je přistupovat k ovládacím prvkům, které tvoří tabulku a nastavovat jim události a vlastnosti. Jednotlivým buňkám je možné nastavovat vlastnosti *Value* a *Style*, pomocí kterých můžeme manipulovat s obsahem a vzhledem buňky. Samotné *DataGridView* má velké množství událostí, jež jsou vyvolány změnou obsahu a stavu buněk a akcemi uživatele. Ne vždycky to ale stačí, například pokud chceme přistupovat k vlastnostem a událostem specifických pro daný ovládací prvek. Můžeme toho dosáhnout pomocí události **EditingControlShowing** třídy *DataGridView*, která nastane, když uživatel rozklikne nějakou buňku. V parametrech delegáta této události je i reference na prvek, jenž tvoří buňku a kterému můžeme pak nastavovat vlastnosti a jeho událostem registrovat metody. Aby nedošlo k tomu, že zaregistrujeme pro události jednu metodu vícekrát při opětovném volání události *EditingControlShowing*, zrušíme nejdřív její registraci a tím docílíme, že metoda je zaregistrována právě jednou.

Jak můžeme vidět na výpise 5, my registrujeme metody do událostí **KeyPress**, **KeyDown** a **TextChanged** *TextBox*u. Pomocí *KeyPress* zabraňujeme, aby stisknutá klávesa nevytvořila chybnou permutaci. Jelikož text popisující permutace obsahuje pouze číslice od jedné do hodnoty *Length*, všechny ostatní klávesy kromě *Backspace* ignorujeme. I přesto je klávesa *Delete* stále

zaregistrována, maže text a vznikají nám neplatné permutace. Klávesa *Delete* z mě neznámého důvodu vůbec nezpůsobí událost *KeyPress* a proto je stále zaregistrována. Vyvolá však už metodu *KeyDown*, kde už můžeme její vyhodnocení zastavit.

```
tb.KeyPress -= Tb_Key_Press;
tb.KeyPress += Tb_Key_Press;
tb.KeyDown -= Tb_KeyDown;
tb.KeyDown += Tb_KeyDown;
tb.TextChanged -= Tb_TextChanged;
tb.TextChanged += Tb_TextChanged;
```

Výpis 5: Zabránění vícenásobné registrace metody do události

Ovšem stále se může stát, že buňky budou obsahovat chybné permutace, pokud už máme nějaké permutace vyplněny a následně snížíme hodnotu *Length*. Pak můžou buňky obsahovat moc velká čísla. Proto když dojde ke změně *Length*, kontrolujeme permutace a pokud některá obsahuje chybné číslo, obarvíme text její buňky na červeno. Při vyvolání události *TextChanged*, je kontrolováno, zda nebyl text opraven a neměli bychom ho obarvit zpátky na černo. Tady jsem se však setkal s trochu nečekaným chováním. Když text buněk obarvuji na červeno, tak nastavuji vlastnost buňky **Style.ForeColor**, ale v *TextChanged* nemám přístup k vlastnostem buňky. Měním proto vlastnost *TextBox*u **ForeColor**. Když jsem měnil *Style.ForeColor* buňky, došlo také ke změně vlastnosti *ForeColor TextBox*u, ale naopak to neplatí. Takže i když během editace *TextBox*u text změnil barvu na černou, tak po ukončení editace se text obarvil zpátky na červeno a i vlastnost *ForeColor TextBoxu* byla nastavena zpátky na červenou. Vyřešil jsem to událostí *DataGridView.CellValueChanged*, která, jak už jméno napovídá, nastane po změně obsahu buňky. V ní tedy nastavuji *Style.ForeColor* buňky na správnou hodnotu. Díky kombinaci událostí *TextBox.TextChanged* a *CellValueChanged* jsem tedy schopen změnit barvu textu jak během editace, tak po jejím ukončení.

Tabulka umožňuje také kopírování obsahu buněk, řádků a sloupců. Okno si proto ukládá při kopírování hodnoty označených buněk do listu textových řetězců a při vkládání kopíruje hodnoty z listu do označených buněk. Trochu problém je, že kolekce uložená ve vlastnosti **SelectedCells** *DataGridView* v případě označení řádku nebo sloupce obsahuje i buňky, které nejsou zobrazeny. Protože nechci vůbec pracovat s nezobrazenými buňkami kopíruji z nebo vkládám do tolika buněk, kolik jsou hodnoty *Layer* pro sloupce a *Options* pro řádky. Stejným způsobem jsou obsahy buněk vymazávány.

Poslední funkcí je automatické zvětšování tabulky se zvětšováním okna. Přitom se automaticky zvětšují rozměry buněk a velikost písma. *DataGridView* umožňuje automatickou změnu šířky sloupců, ale novou výšku řádků musíme vypočítat a nastavit sami. Abychom získali výšku řádku, jednoduše podělíme výšku tabulky bez výšky hlaviček sloupců maximálním počtem vrstev (pro nás 10). Velikost písma se počítá jako $vyskabunky/3.5$ a kontroluje se, zda je buňka

dost široká pro text permutace s maximálním počtem prvků (v našem případě 8 číslic). Pokud je text příliš široký, je zmenšen tak, aby se vešel i na šířku.

6.2.3 Tlačítka

Tlačítko **Randomize** využívá statickou funkci třídy *Game* (Sekce 4.3) a generuje hru na základě hodnot *Length*, *Options* a *Layers*. Při stisknutí tlačítka **Fix inputs** se prochází obsah všech zobrazených buněk a pokud buňka obsahuje aspoň nějaká čísla, tak je její obsah rozdělen na podřetězce obsahující jednotlivá čísla. Z podřetězců, jež obsahují číslo menší než hodnota *Length*, je za pomocí *StringBuilder*u sestaven nový řetězec, který je vložen zpátky do buňky.

Po stisknutí tlačítka **Confirm** se okno snaží vytvořit novou hru ze zadaných dat. Případné chyby ukládáme do listu textových řetězců, jež obsahují informace o důvodech vzniku chyb. Postupně vytváříme instance třídy *Permutation* (Sekce 4.1), pomocí konstruktoru s parametry *string* a *int*, kde *string* je text buňky. Pomocí *try-catch* bloku odchytáváme výjimky, které konstruktor vyhazuje a zapisujeme chyby. Postupně procházíme všechny zobrazené buňky a prázdné ignorujeme, avšak všechny zobrazené řádky musí mít aspoň jednu buňku vyplněnu, pokud ne zaznamenáme chybu. Pro každý řádek si ukládáme instance *Permutation* do listu, pomocí kterého vytvoříme instanci třídy *Layer* (Sekce 4.2), již uložíme do listu vrstev. Stejně jako buňky zkontrolujeme i *Result* a kontroluje se, zda je vyplněno pole *Name* a zda je aspoň jedna vrstva nestatická. Pokud není list s chybami prázdný, je zobrazen *InfoForm* s informacemi o chybách. Pokud nebyly žádné chyby nalezeny, je vytvořena nová instance třídy *Game* (Sekce 4.3), kterou si uložíme do veřejné vlastnosti **NewGame** okna, a následně na ni zavoláme metodu *FindSolutions*. Kontrolujeme, jestli metoda vrátila aspoň jeden výsledek a pokud ne, zobrazíme *InfoForm* s chybovou hláškou. Pokud má hra aspoň jedno řešení, zobrazíme *InfoForm* s informacemi o hře a seznamem řešení. Pokud v *InfoFormu* uživatel potvrdí vytvoření hry, je *CreateGameForm* zavřen a pomocí *NewGame* může *FormMenu* (Sekce 5.1) přistupovat k této nové hře.

6.3 InfoForm

Reprezentuje informační okno, který vzniká a zaniká společně s instancí *CreateGameForm* (Sekce 6.2). Pokud se uživatel pokusí okno zavřít, pouze se schová a zobrazí se při novém stisku tlačítka *Confirm* v *CreateGameForm*. Při zvětšování se zvětšuje i *TextBox*, který je *GameRichTextBox*em (Sekce 6.1.4), tedy mění velikost textu společně se svojí velikostí.

6.3.1 Tlačítko Create

Tlačítko *Create* potvrzuje vytvoření hry. Po jeho stisknutí se nastaví vlastnost **DialogResult** okna *CreateGameForm* na *DialogResult.OK*, což informuje *FormMenu* (Sekce 5.1) o nutnosti aktualizovat seznam her o hru uložené ve vlastnosti *NewGame*.

6.4 FormGame

FormGame je okno, ve kterém hrajeme hru. V konstruktoru je předána hraná hra a většina ovládacích prvků je vytvořena na základě vlastností této hry. Okno obsahuje panel s hrou, panel s posledním výsledkem, tlačítka pro nastavování vstupu, tlačítko nastavení, tlačítko *Start* a tlačítko *Reset*. Pro potřeby tohoto okna jsem vytvořil několik tříd.

6.4.1 GamePanel

Nejdůležitější prvek je **GamePanel** (rozšíření třídy *Panel*), na který je hra kreslena. *GamePanel* používá *DoubleBuffering*, aby zabránil blikání při rychlém překreslování panelu.

6.4.1.1 GamePanelVariables Tato třída obsahuje hodnoty, které se využívají při vykreslování hrací plochy a každý *GamePanel* má na jednu instanci uloženou referenci. Důvodem jejího vzniku byla potřeba tyto hodnoty přepočítávat při změně velikosti panelu. Má 6 veřejných vlastností. Vlastnosti **PermutationSize** a **Spaces** jsou typu **Size** (knihovní struktura), která má vlastnosti **Width** a **Height**. *PermutationSize* určuje velikost obdélníků a *Spaces* velikost mezer mezi nimi. *Spaces* také určují velikost mezer mezi okraji panelu a hrou samotnou. **LineSpace** určuje vzdálenost mezi čárami, **BallRadius** je poloměr kruhů, **SelPermX** reprezentuje x souřadnici levé hrany obdélníků vybraných permutací a **ResBallY** je y souřadnice kružnic výsledku. Má dva konstruktory. První s parametry rozměrů obdélníků a mezer a referencí na hru. Rozměry jednoduše přiřadíme a ostatní proměnné dopočítáme v metodě *InitVars(Game)*, která je ukázaná na výpise 6.

```
private void InitVars(Game game)
{
    LineSpace = PermutationSize.Width / (game.PermLength + 1);
    BallRadius = LineSpace * 2 / 5;
    SelPermX = (game.MaxLayerLength - 1) * (PermutationSize.Width +
        Spaces.Width) + Spaces.Width;
    ResBallY = game.Layers.Count * (Spaces.Height + PermutationSize.Height) +
        Spaces.Height * 2;

    if (BallRadius > Spaces.Height * 3 / 5)
        BallRadius = Spaces.Height * 3 / 5;
}
```

Výpis 6: Metoda InitVars

Druhý konstruktor má jako parametry jinou instanci *GamePanelVariables*, instanci *Game* a poměry (*floaty*) výšek a šířek. Účelem tohoto konstruktoru je vytvořit novou instanci na základě instance jiné vynásobením její hodnot předanými poměry. Vynásobit nám stačí pouze

PermutationSize a *Spaces*, zbytek dopočítáme pomocí *InitVars*. Tento konstruktor se využívá při počítání nových rozměrů po změně velikosti panelu.

6.4.1.2 Animace Úkolem panelu je hru animovat. Potřebujeme instanci třídy **Timer**, která má událost **Tick**, jež je vyvolána pokaždé, když uběhne určený časový interval. Při nastání této události, překreslujeme panel. Během animace vlastně neděláme nic jiného, než že měníme souřadnice, na kterých vykreslujeme kruhy. Panel má následující proměnné:

ballY Y souřadnice kruhů uložená jako float.

ballsX Pole x souřadnic (*float*) kruhů.

nextY Další y souřadnice, kdy kruhy vstoupí nebo vystoupí z permutace.

currLayer Index vrstvy jejíž permutací kruhy právě prochází.

xChange *Boolean* hodnota určující, zda jsou kruhy v obdélníku permutace.

ballSequence Pole *int* hodnot, vyjadřující pořadí kruhů před vstupem do permutace (kruhy jsou číslovány od 1).

nextSequence Pole *int* hodnot, vyjadřující pořadí kruhů po výstupu z permutace.

ballsYChange *Float* hodnota vyjadřující změnu y souřadnice kruhů.

ballsXChange Pole *float* hodnot, vyjadřující změny x souřadnic kruhů během průchodu permutací pro $ballsYChange = 1$.

Pro zjednodušení počítáme s tím, že *ballsYChange* je rovno jedné. Před každým překreslením zvětšíme y souřadnici o *ballsYChange* a pokud je $xChange = true$ (kruhy jsou uvnitř permutace), pak taky x souřadnice o hodnoty uložené v *ballsXChange*. Pokud $ballY$ potom dosáhlo hodnoty $nextY$, provádíme další výpočty podle toho, zda kruhy vstupují do nebo vystupují z permutace. Pokud kruhy vstupují zvětším $nextY$ o výšku permutace a zjistím *nextSequence*. Poté pomocí *ballSequence*, *nextSequence*, *LineSpace* a *PermutationSize.Height* (GamePanelVariables) jsem schopen vypočíst nové hodnoty pro *ballsXChange*, jak je vidět na výpise 7.

Pokud kruhy vystupují z permutace, tak zvětším $nextY$ o velikost vertikální mezery, inkrementuji *currLayer* a do *ballSequence* přiřadím *nextSequence* (výpis 8).

Jakmile *currLayer* dosáhne hodnoty *Layers.Count* hry a zároveň $ballY$ dosáhlo hodnoty $nextY$, znamená to, že kruhy dosáhli kružnic dole a animace je u konce. Potom na instanci naší hry zavoláme metodu *EvaluateResult* s parametrem *ballSequence* a pokud vrátí *true* našli jsme správný vstup a vyřešily hru.

Toto je ale trochu zjednodušené, protože jsme počítali s tím, že *ballsYChange* je rovno jedné. Doopravdy však nabývá i větších hodnot podle toho, jak veliký je panel a jak rychlá je animace.

```
nextSequence = game[currLayer].Permutate(ballSequece);
nextY += GPV.PermutationSize.Height;

for (int i = 1; i <= game.PermLength; i++)
{
    int startIndex = Array.IndexOf(ballSequence, i);
    int endIndex = Array.IndexOf(nextSequence, i);
    float distance = (endIndex - startIndex) * GPV.LineSpace;
    ballsXChange[i - 1] = distance / GPV.PermutationSize.Height;
}
```

Výpis 7: Kruhy vstupují do permutace

```
ballSequece = nextSequence;
currLayer++;
nextY += GPV.Spaces.Height;
```

Výpis 8: Kruhy vystupují z permutace

Jelikož *ballXChange* je změna hodnoty x pokud je *ballYChange* rovno jedné, musíme tyto hodnoty před přičtením vynásobit *ballYChange*. To není nic složitého, ale změna *ballYChange* má i jiný následek. Jedná se o to, že *ballY* může po přičtení přesáhnout hodnotu *nextY*. Musíme tedy kontrolovat, zda není hodnota *ballY* větší než *nextY*, což opět není problém, ale problém už je, že v tom případě jsou x souřadnice kruhů chybné. Například kruhy jsou v permutaci, *ballYChange* je 3 a *ballY* je o 1 menší než *nextY*. Při další inkrementaci *ballY* bude *ballY* o dvě větší než poslední *nextY* a my přičetli *ballsXChange* třikrát místo jednou, jak by to mělo být správně. Kruhy tedy nebudou ležet na čarách. Podobný problém nastává i při vstupu do permutace, kdy kruhy naopak nebudou měnit x, když by už měli. Vyřešil jsem to tak, že pokaždé při překročení hodnoty *nextY* zjišťuji o kolik byla tato hodnota překročena, potom spočítám o kolik se x souřadnice odchýlily a tyto hodnoty pak přičtu k x souřadnicím a tím je dorovnám.

6.4.2 LastResultPanel

Okno obsahuje, ještě další typ odvozený od *Panel*u a tím je *LastResultPanel*. Je umístěn pod *GamePanel*em a jak jméno napovídá zobrazuje výstup posledního testovaného vstupu. Má jeden konstruktor, ve kterém mu je předána reference na hru. Má čtyři vlastnosti. **Spaces** je velikost horizontálních mezer mezi kruhy, **BallDiameter** reprezentuje průměr kruhů a **FirstX** je x souřadnice prvního kruhu. Tyto vlastnosti si vypočítává panel sám na základě svých rozměrů a vlastností hry. Vlastnost **LastResult** je permutace, která byla posledním výsledkem a pomocí této vlastnosti nastaví okno zobrazovaný výsledek po každém vyhodnocení hry.

6.4.3 Tlačítka určena k ovládání hry

Tlačítka se šipkami jsou tvořená dynamicky podle počtu nestatických vrstev. Jejich smyslem je měnit hodnotu vlastnosti *SelPerm* jednotlivých vrstev pomocí metod *SelectRight* a *SelectLeft* (Sekce 4.2) a měnit tak vstup permutačního programu. Aby se dalo při kliknutí určit, ke které vrstvě tlačítko patří a kterým směrem jí tlačítko hýbe, vytvořil jsem třídu **MoveButton**, která dědí z třídy *Button*. Má navíc vlastnosti **IsLeft**, říkající jestli je tlačítko nalevo od *GamePanel*u, a vlastnost **Layer** obsahující index vrstvy, kterou tlačítko manipuluje.

6.4.4 Vstupy klávesnice

Zde je nutno zmínit, že i když při stisku klávesy je vyvolána událost *KeyDown* okna, pomocí které můžeme reagovat na jednotlivé klávesy, tak skutečnost je taková, že pro *šipky* a *Enter* to tak úplně nefunguje. Okno má totiž pro tyto klávesy předdefinované chování. Pomocí *šipek* se vybírá právě aktivní prvek okna a na *Enter* reaguje aktivní prvek svým předdefinovaným chováním. Stisknutí těchto kláves pak ani nevyvolává událost *KeyDown* okna. Abychom mohli stisknutí těchto kláves registrovat, musíme všem ovládacím prvkům okna zaregistrovat do jejich události *PreviewKeyDown* metodu, ve které nastavíme tyto klávesy jako vstupní klávesy. Potom ztratí tyto klávesy svoje předchozí funkce a konečně způsobují vyvolání události *KeyDown*.

6.5 GameSettingsForm

Okno, které umožňuje měnit nastavení hry. Okno nelze zavřít. Křížek, tlačítko *OK* nebo klávesa *Escape* pouze schová okno. Toto okno může existovat v aplikaci pouze jednou, použijeme proto návrhový vzor **Singleton** (jedináček). Neexistuje tedy žádný veřejný konstruktor a uchováváme si statickou instanci třídy, která je buď *null* nebo obsahuje referenci na už vytvořenou instanci. Statickou metodou *Open()* buď vytvoříme a otevřeme nové okno, pokud je instance *null*, v ostatních případech zobrazíme už to existující.

6.5.1 GameSettings

Tato třída reprezentuje nastavení hry. Toto nastavení chceme v celé aplikaci pouze jednou, opět proto použijeme návrhový vzor **Singleton**. Třída má statickou vlastnost **Instance**, ve které je uložena ona jediná instance. Je veřejná pro čtení, můžeme k ní tedy kdykoliv přistupovat. Třída opět nemá jediný veřejný konstruktor a tak zabráníme vytvoření další instance.

GameSettings má tři veřejné vlastnosti. Vlastnost **AnimationSpeed** typu *int* vyjadřuje o kolik se mění y souřadnice kruhů (proměnná *ballsYChange*, Sekce 6.4.1.2) během animace *GamePanel*u (Sekce 6.4.1). Hodnota této vlastnosti tedy určuje, jak rychlá je animace. Další vlastnost je **AnimationEnabled** typu *bool*, pomocí ní můžeme animaci úplně vypnout. Kruhy se při stisknutí tlačítka *Start* v okně hry (Sekce 5.4) vůbec nepohnou a jenom se překreslí *LastResultPanel* (Sekce 6.4.2), aby zobrazil výsledek. Poslední vlastností je **LineColor** typu

LinesColor. *LinesColor* je výčtový typ (*enumerable*), který může nabývat hodnot **None**, **Dynamic** a **Full**. Tato vlastnost určuje, jak se mají obarvovat čáry, po kterých se kruhy pohybují. Kreslit barevné čáry je poněkud složitější, protože potřebujeme zjistit, kterému kruhu daná čára patří, tedy jakou má mít barvu. Zjišťuji to tak, že když kreslím čáry, tak znám počáteční stav kruhů a pak beru permutaci po permutaci, jak jimi kruhy prochází a zjišťuji změnu jejich pořadí a na základě těchto údajů kreslím barevné čáry. Funguje to vlastně podobně jako při počítání změn x souřadnic při animaci. Mám dvě sekvence reprezentující pořadí před a po průchodu permutací. Pomocí nich zjistím počáteční a konečné body barevných čar. U dynamických čar ještě musím končit pozicemi kruhů, což už není takový problém, protože jejich x a y souřadnice znám.

Nastavení uložené ve vlastnosti *Instance* ukládám při ukončení aplikace. Při otevření je opětovně nahráno a instance je vytvořena pomocí statické metody **Load**, do které se předává object získaný pomocí *FileManager*u (Sekce 4.4). Poslední důležitá věc je událost **SettingsChanged**, jež je vyvolána pokud došlo ke změně *LinesColor* nebo *AnimationSpeed*. Při vytvoření instance *FormGame* je do této události zaregistrována metoda **UpdateSetts** *GamePanel*u (Sekce 6.4.1), jež nastaví potřebné hodnoty a vyvolá okamžitou změnu dokonce i za běhu animace.

6.5.2 TrackBarInterval

TrackBar je prvek, který umožňuje vybírat celočíselnou hodnotu z daného intervalu. Tento *TrackBar* mění hodnotu vlastnosti *AnimationSpeed* instance nastavení. Interval hodnot ze kterých je možno vybírat je $\langle 1, 5 \rangle$. 5 jsem určil, jako vhodnou maximální hodnotu, protože pohyb kruhů nebude vypadat skokově a navíc je to půlka základní délky vertikální mezery, která je 10. Je důležité, aby to bylo méně než 10, protože pak by kruhy mohli mezery mezi permutacemi úplně přeskočit a přestala by nám fungovat animace, jelikož počítáme s tím, že aspoň jednou vykreslíme kruhy i v mezerách. Další zajímavost je, že okno má vlastní *Timer*, jež je pokaždé spuštěn znovu při vyvolání události **ValueChanged** *TrackBar*u. Až po uplynutí intervalu *Timer*u, je nastavena hodnota *AnimationSpeed*. To je z toho důvodu, aby nebyla hodnota zbytečně měněna pouhým tažením přes hodnoty, ale pouze pokud je posuvník ponechán na hodnotě delší dobu (0.5 s).

7 Ukázková kolekce permutačních programů

Posledním úkolem je vytvořit ukázkovou kolekci permutačních programů, na kterých by se dala hra testovat. Hry mají stejně jako permutační programy (Sekce 2.4) několik vlastností: počet vstupů, počet možných výstupů, počet vrstev, maximální velikost vrstev a počet prvků permutací. Samozřejmě čím vyšší jsou tyto hodnoty, tím více bude hra složitá. Největším ukazatelem složitosti je ale poměr možných vstupů ku počtu vstupů, které jsou řešeními. Jakmile více jak 5% vstupů vrací správné řešení, je hra snadná, mezi $1-5\%$ je hra středně těžká a pro méně než 1% je už hra obtížná.

7.1 Ukázkové hry

1. **Easy game**: Jedná se o ukázku jednoduché hry s jedním řešením a 8 různými vstupy. Řešením je $12,5\%$ vstupů.
2. **Normal game**: Jedná se o ukázku hry normální obtížnosti s jedním řešením a 36 různými vstupy. Řešením je $2,778\%$ vstupů.
3. **Hard game**: Jedná se o ukázku hry těžké obtížnosti s 11 řešeními a 5184 různými vstupy. Řešením je $0,212\%$ vstupů.
4. **Game 4**: Hra zobrazena na Obrázku 6. Jednoduchá hra. Má 2 řešení a 32 různých vstupů. Řešením je $6,25\%$ vstupů.
5. **Example**: Hra, jejíž permutační program byl použit jako příklad v Sekci 2.4 a jež je zobrazena na Obrázku 4. Jedná se o jednoduchou hru se 12 různými vstupy a 1 řešením. Řešením je $8,333\%$ vstupů.
6. **My game**: Hra, která je zobrazena na Obrázcích 7, 8 a 9. Středně obtížná hra s 1 řešením a 48 různými vstupy. Řešením je $2,083\%$ vstupů.
7. **All layer sizes**: Hra s vrstvami všech možných velikostí. Má 2 řešení a 2880 možných vstupů. Těžká hra, $0,069\%$ vstupů je řešením.
8. **Max layers**: Hra s maximálním počtem vrstev se dvěma permutacemi s maximálním počtem prvků. Má 1 řešení a 1024 možných vstupů. Těžká hra, $0,098\%$ vstupů je řešením.
9. **Random game**: Náhodně vygenerovaná hra s 8 vrstvami až velikosti 4 a s permutacemi o 7 prvcích. Jedná se o těžkou hru s 4608 možnými vstupy a dvěma řešeními. $0,043\%$ vstupů je řešením.
10. **Max test**: Hra s maximálním počtem vrstev, jež jsou všechny maximální velikosti a obsahují permutace s maximálním počtem prvků. Testuje zda aplikace zvládá hru maximální velikosti. Má 223 řešení a 9765625 možných vstupů. Velice obtížná hra, pro kterou pouze $0,002\%$ vstupů je řešením.

8 Závěr

V rámci práce byl vytvořen program, který umožňuje tvorbu, editaci a experimentování s permutačními programy, což bylo naším zadáním. Snažil jsem se udělat uživatelské rozhraní co nejvíc intuitivní, jednoduché a zároveň s funkcemi, které by mohl uživatel požadovat a zjednoduší mu práci. Nutno podotknout, že jsem se v tomto textu nezabýval implementací uživatelského rozhraní do podrobností, ale snažil jsem se hlavní problémy a jejich řešení zmínit. Pokud bych měl zhodnotit, co bylo nejsložitější naimplementovat, tak by to bylo určitě vykreslování hry, její animování a zvětšování oken s jejich prvky. Aplikace by se samozřejmě dala ještě vylepšit. Jedním vylepšením by mohlo být přidaní možnosti tvoření her pomocí grafického rozhraní podobného samotné hře. Uživatel by pak už při tvorbě viděl, jak bude hra vypadat. Abych řekl pravdu, tak mě samotného implementace bavila a měl jsem radost, když jsem viděl, jak se program rozšiřuje o více a více funkcí, které fungovaly, tak jak jsem si představoval. Nemůžu však na sto procent zaručit bezchybovost programu, což se domnívám nelze u žádného programu. Program jsem samozřejmě průběžně testoval, ať už jeho jednotlivé funkce stejně tak jako celek. Ale jak říkal pán Ing. Tomáš Gregor na jeho přednáškách týkajících se vývoje informačních systémů, chyby v programu se dělí na objevené a neobjevené. Já o žádných objevených nevím, ale možnost existence neobjevených nelze vyloučit.

Literatura

[1] KUBESA, Michael. *Základy diskrétní matematiky* [online]. Ostrava: FEI VŠB - TUO, 2012 [cit. 2019-04-24]. Dostupné z: http://homel.vsb.cz/ kov16/files/zaklady_diskretni_matematiky.pdf

[2] BARRINGTON, David A. Bounded-width polynomial-size branching programs recognize exactly those languages in NC1. *Journal of Computer and System Sciences* [online]. 1989, **38**(1), 150-164 [cit. 2019-04-26]. DOI: 10.1016/0022-0000(89)90037-8. ISSN 00220000. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/0022000089900378

[3] HERTRAMPF, U., C. LAUTEMANN, T. SCHWENTICK, H. VOLLMER a K.W. WAGNER. On the power of polynomial time bit-reductions. *[1993] Proceedings of the Eigth Annual Structure in Complexity Theory Conference* [online]. IEEE Comput. Soc. Press, 1993, , 200-207 [cit. 2019-04-26]. DOI: 10.1109/SCT.1993.336526. ISBN 0-8186-4070-7. Dostupné z: http://ieeexplore.ieee.org/document/336526/

[4] KENNEDY, John a Michael SATRAN. GDI+. *Microsoft Docs* [online]. [cit. 2019-04-24]. Dostupné z: https://docs.microsoft.com/en-us/windows/desktop/gdiplus/-gdiplus-gdi-start

[5] System.Windows.Forms Namespace. *Microsoft Docs* [online]. [cit. 2019-04-24]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/api/system.windows.forms?view=netframework-4.8

[6] System.Drawing Namespace. *Microsoft Docs* [online]. [cit. 2019-04-24]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/api/system.drawing?view=netframework-4.7.2

A Příloha v IS EDISON

Příloha obsahuje následující složky a soubory:

2019_CHO0163_BP.pdf

Digitální kopie této práce ve formátu PDF/A.

Složka Aplikace

Tato složka obsahuje samotnou aplikaci s následujícími soubory:

Game with permutation programs.exe Spustitelný soubor aplikace. Pro spuštění aplikace je vyžadován operační systém Windows či nainstalovaný Mono framework.

PermutationPrograms.dll Dynamická knihovna s třídami permutačních programů potřebných pro spuštění aplikace

Game with permutation programs.exe.config Konfigurační soubor aplikace.

games.bin Do tohoto souboru si aplikace ukládá vytvořené hry, které jsou nahrány při jejím dalším otevřením. Před prvním otevřením aplikace obsahuje soubor ukázkové hry.

settings.bin Tento soubor není před prvním spuštěním ve složce přítomen, ale bude vytvořen po prvním zavření aplikace. Obsahuje nastavení hry, která si uživatel v aplikaci může navolit a při dalším spuštění aplikace jsou načtena.

PresetGames.bin Tento soubor není používaný aplikací, ale obsahuje kopii ukázkových her. V případě potřeby spustit hru s načtením původních ukázkových her, potom co byli ty ze začátku uložené v *games.bin* editovány či smazány, lze toho docílit zkopírováním tohoto souboru a přejmenováním kopie na *games.bin*. Samozřejmě je nutno původní *games.bin* soubor, přejmenovat, přesunout nebo smazat.

Složka Visual Studio projekt

Obsahuje zdrojové soubory, z nichž byla sestavena aplikace. Obsahuje **Game with permutation programs.sln**, jež otevře projekty ve *Visual Studiu*. Je vyžadováno *Visual Studio 2017* či novější. Ve *Visual Studiu* lze zdrojové kódy zkompilovat a aplikaci sestavit.